TECHNISCHE UNIVERSITÄT DARMSTADT
HOCHSCHULKENNZIFFER D-17
DEPARTMENT OF COMPUTER SCIENCE
SECURITY ENGINEERING GROUP

PhD Thesis

Secure Computations on Non-Integer Values

Dissertationsschrift

**in englischer Sprache
zur Erlangung des Grades eines Dr. rer. nat.
an der Technischen Universität Darmstadt**

Dipl.-Math. Dipl.-Inform. Martin Franz
geboren 14.12.1982 in Alzenau (Bayern)

Erstreferent:	Prof. Dr. Stefan Katzenbeisser
Korreferent:	Prof. Dr. Somesh Jha

Tag der Einreichung: 20.9.2011
Tag der Prüfung: 18.11.2011

Darmstadt 2011

Abstract

Research in the scientific field of *Secure Multiparty Computation* (SMC) was started in the year 1982, when Andrew C. Yao presented the well known Millionaires' problem [1]. Since then, this area of research has witnessed many new theoretical results and technological advances, which made it possible to realize a large scale of applications using techniques from SMC. However, one class of applications has barely been touched in the past 25 years: The question of how to perform secure computations with non-integer values, e.g. with values in a floating point representation on large intervals taken from the domain of real values. This thesis presents new results in this field of research.

Our first contribution is a new computational framework for SMC with real values which are stored in a logarithmic encoding. This includes on the one hand a representation scheme which allows to encrypt values stored in this representation. On the other hand, our computational framework consists of secure protocols that allow to perform all arithmetic operations on encrypted values encoded in this representation scheme.

Next, we present a secure and efficient implementation of the IEEE 754 floating point standard. Our approach shows how to encrypt floating point values in a way that all arithmetic operations, including the normalization operation, can be performed interactively between two mutually distrusting parties with acceptable computational and communication overhead.

We round off this dissertation with both a theoretical and a practical evaluation of the proposed techniques. Firstly, we give a thorough theoretical complexity analysis which compares the two approaches in terms of computational and communication complexity. We further compare the two different approaches with a basic scheme for a fixed point representation, showing that both approaches indeed outperform the fixed point representation for computational problems of typical size.

Finally, we demonstrate the practical importance of the presented techniques.

For this, we implemented two important algorithms from bioinformatics in the developed framework, namely the forward and Viterby algorithm. These algorithms typically tend to be numerically unstable, since they require computations on decreasingly small probabilities. Our implementation shows that, using the theoretical approach presented in this dissertation, it is in fact possible to efficiently solve these real world problems in the encrypted domain.

Zusammenfassung

Die Forschung zu Secure Multiparty Computation (SMC) begann im Jahr 1982, als Andrew C. Yao das Millionärsproblem vorstellte [1]. Seitdem hat die Wissenschaft in diesem Bereich große Fortschritte gemacht, viele sicherheitskritische Anwendungen wurden mittels SMC realisiert. Einzig die Anwendungen, die mathematische Berechnungen auf nicht-ganzzahligen Werten durchführen, waren lange Zeit von diesen Fortschritten ausgeschlossen. Zu diesen Anwendungen gehören beispielsweise Algorithmen, die Berechnungen auf großen Intervallen mit reellen Zahlen durchführen.

Die vorliegende Dissertation präsentiert neue Ergebnisse in diesem Forschungsbereich. Zunächst wird eine neue Methode vorgestellt, die es erlaubt, sichere Berechnungen auf reellen Zahlen durchzuführen, die in einer logarithmischen Repräsentierung gespeichert sind. Zum einen wird beschrieben, wie so repräsentierte Zahlen effektiv verschlüsselt werden können. Danach werden kryptographische Protokolle angegeben, die es erlauben bestimmte arithmetische Operationen mit auf diese Weise kodierten und verschlüsselten Werten durchzuführen.

In einem weiteren Kapitel wird eine sichere Umsetzung des IEEE 754 Gleitkommastandards präsentiert. Diese zeigt auf, wie Gleitkommazahlen verschlüsselt werden können. Zudem werden kryptographische Protokolle beschrieben, die es erlauben Berechnungen auf solch verschlüsselten Gleitkommazahlen durchzuführen.

Abgeschlossen wird diese Dissertation mit sowohl einer theoretischen, als auch einer praktischen Evaluierung der hier vorgestellten Techniken. Zunächst werden in einer ausgiebigen theoretischen Komplexitätsanalyse die Rechen- wie auch die Kommunikationskomplexität der beiden neu vorgestellten Methoden zum Rechnen mit verschlüsselten Zahlen vorgestellt. Danach wird die Performanz dieser beiden Methoden mit einer Standardmethode verglichen, die auf einer Festpunktarithmetik basiert. Es zeigt sich, dass beide Methoden für typische Probleme deutlich effizienter sind als die Festpunktarithmetik.

Zum Abschluss wird auch die praktische Machbarkeit der neu vorgestellten Techniken demonstriert. Dafür wurden zwei wichtige Algorithmen aus der Bioinformatik implementiert, der Forward- und der Viterby Algorithmus. Diese Algorithmen sind typischerweise numerisch instabil, denn sie führen ihre Berechnungen auf ständig kleiner werdenden Wahrscheinlichkeiten durch. Die hier vorgestellte Implementierung zeigt, dass die neuen theoretischen Methoden auch in der Praxis erfolgreich eingesetzt werden können, um real vorkommende Probleme zu lösen.

Acknowledgements

I would like to thank Stefan Katzenbeisser for being my advisor. Thank you for the support and encouragement in these three years at SecEng & CASED.

I am grateful to all people in the SecEng team at TU Darmstadt and the people I met at CASED. In particular I would like to thank Sascha Müller, Martin Mink for helpful comments about my final presentation, Björn Deiseroth and Waqas Sharif for their help and effort implementing the framework. I thank Andrea Püchner and Heike Meissner for their help with big and small things in the past three years.

I would like to thank Nicole Voss and Benoit Libert for proofreading and helpful comments.

Radu Sion was a great host during my research visit in New York. Thank you for all the advice and meetings in the most interesting locations. I would also like to thank Peter Williams for the joint work both here in Darmstadt and in NYC.

I am particularly grateful to Somesh Jha for always having fresh ideas for my research and Kay Hamacher for the introduction to and support with various topics from computational biology.

Finally and most importantly, I would like to thank my parents Anette and Herbert and the rest of my family for their constant help and support during all the years.

Contents

1 Introduction **11**

1.1 Outline . . . 13

2 Preliminaries and Cryptographic Primitives **17**

2.1 Attacker Model . . . 17

2.1.1 Semi-honest Attacker Model . . . 17

2.1.2 Malicious Attacker Model . . . 18

2.2 Homomorphic Encryption . . . 18

2.2.1 Operations in the encrypted domain . . . 20

2.2.2 The Paillier-Encryption Scheme . . . 21

2.2.3 DGK Cryptosystem . . . 25

2.3 Oblivious Pseudo-Random Function Evaluation . . . 27

2.4 Oblivious Transfer . . . 28

2.5 Notation . . . 28

3 Secure Computations on Integer Values **31**

3.1 Garbled Circuits . . . 32

3.2 Secret Sharing . . . 34

3.2.1 Additive Blinding . . . 35

3.2.2 Multiplicative Blinding . . . 35

3.3 Homomorphic Encryption . . . 36

3.3.1 Secure Multiplication . . . 37

3.3.2 Integer Comparison . . . 38

3.4 Related Work . . . 41

3.4.1 SMC on Binary and Integer Arithmetic . . . 41

3.4.2 SMC on Non-Integer Values . . . 42

3.4.3 Selected SMC Applications . . . 42

4 Secure Computations in Fixed Point Representation 45
4.1 General Idea 45
4.2 Rounding Strategies 46
4.2.1 Probabilistic Rounding Protocol 46
4.2.2 Deterministic Rounding Protocol 48

5 Secure Computations using Logarithmic Encoding 49
5.1 Private Function Evaluation 50
5.1.1 OPRF Construction 50
5.1.2 OT Construction 54
5.2 Secure Computations on Non-Integer Values 61
5.2.1 Data Representation 61
5.2.2 Parameters 62
5.2.3 Arithmetic Operations 63
5.2.4 Secure Operations 65
5.2.5 Multiplication protocol 68
5.3 Discussion: Alternative **LSUM** Implementations 69

6 Secure Computations using Floating Point Representation 71
6.1 IEEE 754 Floating Point 71
6.2 Secure Protocols 72
6.2.1 Normalization 73
6.2.2 Subtraction and Addition 75
6.2.3 Multiplication 78
6.2.4 Division 79
6.2.5 Exception Handling 79
6.3 Security 82

7 Discussion 83
7.1 Basic Properties 83
7.2 Complexity Analysis 83
7.2.1 Computational Complexity 84
7.2.2 Communication Complexity 87
7.2.3 Overall Comparison 89

8 Applications 91
8.1 Secure Bioinformatics and HMM Evaluation 92
8.2 Hidden Markov Models 93

8.3 Secure Forward Algorithm 94
8.4 Secure Viterbi Algorithm 96
8.5 Implementation and Experimental Results 99
8.5.1 Complexity of **LSUM** 100
8.5.2 Complexity of private HMM analysis. 102

9 Conclusions and Future Work 107

References 109

Chapter 1

Introduction

The dawn of Secure Multiparty Computation (SMC) dates back to the year 1982, when Andrew C. Yao presented the well known Millionaires' problem [1]. One can say that this started a new era in cryptography. Speaking in general terms, the idea of SMC deals with the problem of how two or more parties can securely evaluate a given function on their private inputs. While the result of the secure computations should be revealed to at least one party, one major goal is that the private inputs of each party remain hidden from all other parties.

Since 1982 and to this date, the scientific field of SMC has witnessed a multitude of new results and has undergone various innovative changes. This resulted in a variety of different approaches to SMC. A remarkable early result in this field has been the completeness theorem given in [2], which states that any computable function can be realized by SMC. Many subsequent publications have improved the practical efficiency of SMC. Consequently, new theoretical results in SMC, as well as the technological advances in computer hardware, led to a number of practical applications built on SMC which allow for secure computations on private data.

In fact, due to the ever increasing amount of digitally stored data, there is a growing need for technical solutions that protect sensitive personal information. In the past, data privacy was mainly assured through procedures, laws or access control policies. However, these protection mechanisms are ineffective once data is outsourced to partially untrusted servers or processed by third parties. To alleviate this problem, special *Privacy Enhancing Technologies* (PETs) have been proposed that allow to keep sensitive data encrypted and compute directly on encrypted values using cryptographic protocols. This strategy allows to control, by design of the protocols, the amount of sensitive data that is leaked to the protocol participants. Commonly, these constructions use techniques from Secure Multiparty Computation, which offer various different approaches to compute securely with data represented as integers.

However, many computational problems require processing of very small non-integer values, some of them in the range of 10^{-1} to 10^{-300} or even smaller. The most prominent examples are applications that require processing of probabilities over large sets of events, such as dynamic programming algorithms, where a large number of probabilities have to be multiplied. Other problems require computing with values taken from a large interval such as $[2^{-100}, 2^{100}]$, in a numerically stable way. Numerical computations for eigenvalue or singular value problems, differential equations or matrix inversion constitute important examples of this class of problems. Efficient solutions to these problems typically work with a number representation which has a constant relative error when representing a real value (in contrast to a constant absolute error, which is the case for a fixed point representation). Currently available solutions for SMC provide rather efficient ways to perform computations on integer values. Implicitly, these tools also allow to compute on rational numbers of a fixed precision (by representing the numerator and denominator separately) and on integers in fixed point representation (by appropriate scaling and quantization). However, these frameworks usually cannot be used for the above-mentioned problems, as they are not well suited to represent both large and very small values at the same time. Furthermore, rounding errors can accumulate once a large number of operations are performed.

In this thesis, we provide for the first time protocols allowing secure computations on non-integer values with a constant relative representation error. In fact, this is an important problem in the field of secure computations; as outlined above, many computational problems demand the ability to compute privately on both large and rather small values at the same time. In this thesis we will show that computations on such approximations of real values are possible, and can be used to build practical solutions to real world problems. We present the most recent advances in this line of research; this includes a secure implementation of the IEEE 754 floating point standard, as well as a computational framework which represents non-integer values using a logarithmic encoding.

We demonstrate the applicability of our approach for computing with non-integer values by considering an important problem in the context of bioinformatics. In bioinformatics, it is commonly believed that advances in sequencing technology will allow to sequence human genomes at low cost in the foreseeable future, effectively paving the route for personalized medicine [3, 4, 5]. Genomic data will be used by healthcare providers to check for disease predispositions or drug intolerances for individual patients. It can be foreseen that a service-based industry will emerge, where special providers offer services to match genomic sequences against models of specific diseases, as well as offering personalized programs of drug application to such patients. In this context, two important security problems arise. First, genomic data must be considered extremely privacy sensitive and should thus be strongly protected from abuse. Second, mathematical models for diseases and the related genomic details are valuable intellectual property of the service provider, which is the basis of their business model. Thus, privacy protection of genomic data and protection of the involved intellectual property should be achieved at the same time. Ideally the computations should be performed obliviously in a way that the parties do not need to disclose their data to each other, but still gain the desired result. In this thesis it is shown how to solve a problem in the aforementioned scenario, with good practical performance.

1.1 Outline

Our work on secure computations on non-integer values is based on various SMC techniques, which in turn use a multitude of cryptographic primitives. We therefore review these cryptographic primitives in Chapter 2 and explain important basic concepts of SMC in Chapter 3. In Chapter 4, we describe how the techniques presented in Chapter 3 can be used to perform secure computations on values stored in a fixed point representation.

The remainder of this thesis is dedicated to the exploration of techniques for SMC operating on non-integer values. Chapter 5 presents a framework that allows secure two-party computations on approximations of values taken from a bounded real domain. The proposed solution shows how to use a quantized logarithmic representation of real values, which allows to represent both very small and very large numbers with bounded relative error. Numbers represented in this way can be encrypted using standard homomorphic encryption schemes. Next, we describe protocols that allow to perform all arithmetic operations on such encrypted values. Chapter 5 is based on the papers

[6] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. Schröder. Secure computations on real-valued signals. In *IEEE Workshop on Information Forensics and Security (WIFS'10).* IEEE Press, Dez 2010.

[7] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. Schröder. Towards secure bioinformatics services. In *Financial Cryptography and Data Security (FC'11)*, 2011.

In Chapter 6, we investigated how to compute with encrypted floating point values. Material in this chapter is based on the paper

[8] M. Franz and S. Katzenbeisser. Processing encrypted floating point signals. In *Proceedings of the 13th ACM Workshop on Multimedia and Security (MM&Sec '11)*, New York, NY, USA, 2011. ACM.

The chapter presents a first solution to the problem of computing with encrypted floating point values that adhere to the IEEE 754 floating point standard. In particular, we present secure and efficient protocols which allow to perform all arithmetic operations on such encrypted values. Furthermore, we show how to enhance these protocols to allow for a basic exception handling.

Chapter 7 compares the two approaches to represent and compute with encrypted non-integer values presented in Chapters 5 and 6 with each other. Furthermore, we analyze the computational- and communication complexity of both approaches and compare it to the basic fixed point representation presented in Chapter 4.

In Chapter 8 we describe applications of the techniques developed in Chapter 5 to the domain of privacy-preserving computations. In particular, we show how to run two important algorithms in the context of data analysis using Hidden Markov Models (HMM), namely the Viterbi and the forward algorithm, in a secure manner. As a basic application, we consider the problem of private sequence analysis from bioinformatics. In the current setting one party knows a protein sequence, where the second party knows a HMM. While the parties learn whether the model fits the sequence, they neither have to disclose the parametrization of the model nor the sequence to each other. Practicality of the two applications is illustrated by experiments on realistic protein sequences and HMM models. Our experiments confirm the theoretic analysis put forward in Chapter 7; despite the huge number of arithmetic operations required to solve the problem, we experimentally show that HMMs with sizes of practical importance can obliviously be evaluated using computational resources typically found in medical laboratories. For example, our experiments show that evaluation of a medium sized HMM takes less than 5 minutes, even though about 450.000 arithmetic operations need to be performed. These applications have been presented in [7] and in

[9] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. Schroeder. Secure Computations on Non-Integer Values with Applications to Privacy-Preserving Sequence Analysis.

Chapter 2

Preliminaries and Cryptographic Primitives

2.1 Attacker Model

In the most general setting, we consider n parties jointly running a cryptographic protocol. We will assume that a potential attacker has corrupted a subset of the participating parties, has access to their data and controls the way they interact with the other parties when running the protocol. We will distinguish between two types of attackers (adversarial behavior) which will be described in this section.

2.1.1 Semi-honest Attacker Model

A *semi-honest* (sometimes also called *honest-but-curious*) adversary tries to gain as much information as possible from messages sent during the protocol, but he follows the protocol faithfully and does not perform active attacks on it. For example he might try to learn something from the messages he receives during the protocol run, but he does not actively operate to corrupt, substitute or drop messages. This attacker model is discussed in detail in [10, Chapter 7.2].

In order to prove that a two-party protocol is secure in the semi-honest attacker model, it suffices to show that the views of both parties can be efficiently simulated [10, Chapter 7.2]. For a protocol Π this is the case if whatever can be learned by a party running the protocol can be efficiently computed from the input and output available to that party. This is summarized by the following definition:

Definition. *We say that a two-party protocol* Π *privately computes a function* f *if there exist probabilistic polynomial time algorithms (PPTAs) denoted by* S_1 *and* S_2*, such that*

$$\begin{aligned} \{S_1(x, f_1(x,y))\}_{x,y} &\equiv^c \{\textsc{View}_1^{\Pi}(x,y)\}_{x,y}, \\ \{S_2(x, f_2(x,y))\}_{x,y} &\equiv^c \{\textsc{View}_2^{\Pi}(x,y)\}_{x,y}, \end{aligned}$$

where x *and* y *refer to the inputs of party* 1 *and* 2*, respectively. The view of party* $i \in \{1,2\}$ *is denoted by* $\textsc{View}_i^{\Pi}(x,y)$*, and* $\equiv^c$ *denotes that the two views are computationally indistinguishable.*

This definition is explained in detail in [10]. Security for larger protocols follows from the composition theorem for the semi-honest model [10, Theorem 7.3.3].

2.1.2 Malicious Attacker Model

The second attacker type is referred to as *malicious* adversary. A malicious (sometimes also called *active*) attacker tries to influence the execution of the protocol to achieve his goals. Besides attempting to learn secrets of the other parties, he may try to prevent the protocol from terminating with a correct result or send corrupted information to influence the results of the protocol. It is well known how to transform any protocol secure in the semi-honest attacker model into one that is secure against active adversaries [10, Chapter 7.4]. This is achieved by forcing the malicious attacker to behave in a semi-honest manner. Protocols secure in the semi-honest attacker model are typically much more efficient than protocols that are secure against active adversaries. More details about this attacker type can be found in [10, Chapter 7.4].

2.2 Homomorphic Encryption

Many efficient cryptographic protocols make use of a special class of asymmetric encryption schemes, namely homomorphic encryption. Speaking in mathematical terms, this means that the decryption function

$$D : C \to M$$

that maps a ciphertext $c \in C$ to a plaintext $D(c) = m \in M$ is a homomorphism of groups. Thus, D maps the neutral element of C to the neutral element of M, and for two ciphertexts $c_1, c_2 \in C$ we have that

$$D(c_1) *_1 D(c_2) = D(c_1 *_2 c_2). \tag{2.1}$$

In the typical examples of homomorphic cryptosystems the operation $*_1$ will be replaced by an addition while the $*_2$ stands for the multiplication in the underlying algebraic structure. These cryptosystems are referred to as *additively homomorphic cryptosystems*. We will see concrete examples of cryptosystems with an homomorphic property later in this section.

Note that this property also implies that the encryption

$$E : M \to C$$

that maps a plaintext $m \in M$ to a ciphertext $c \in C$ has some homomorphic properties, namely that an encryption $E(m_1 *_1 m_2)$ can be computed from $E(m_1)$ and $E(m_2)$ by computing $E(m_1) *_2 E(m_2)$, for the same operations $*_1$ and $*_2$ as in Equation (2.1). This can be seen as follows. Let $m_1, m_2 \in M$ two plaintexts and $c_1 = E(m_1)$, $c_2 = E(m_2) \in C$ two probabilistic encryptions of m_1 and m_2. Then, by Equation (2.1), the value $E(m_1) *_2 E(m_2)$ will be decrypted to

$$D(E(m_1) *_2 E(m_2)) = D(c_1 *_2 c_2) \overset{(2.1)}{=} D(c_1) *_1 D(c_2) = m_1 *_1 m_2,$$

and therefore $E(m_1) *_2 E(m_2)$ is an encryption of $m_1 *_1 m_2$. We will frequently use homomorphic encryption, as it allows to perform linear operations directly on ciphertexts.

We further require the cryptosystem to be *semantically secure*. In a semantically secure cryptosystem, it is hard to decide which of two given ciphertexts encrypts a certain plaintext. Vice versa, given a ciphertext and two plaintexts, it is hard to decide which of the two plaintexts is encrypted by the ciphertext. For this reason, semantic security implies *probabilistic encryption* [11]. This requires the encryption function to be probabilistic.

One long time outstanding problem was the question whether so called *fully homomorphic* cryptosystems exist. This question was answered affirmatively in [12]. A fully homomorphic cryptosystem preserves the full ring structure of the plaintext space, i.e. it is homomorphic under both addition and multiplication at the same time. Unfortunately, even though the scheme in [12] has the aforementioned properties, it is far from being used in practice. This is due to the overwhelming costs which are required for the basic operations of the encryption scheme, e.g. the size of a ciphertext or a key is hundreds of megabytes, making the scheme impractical to use in real world applications. Even though the scheme presented in [12] was improved in [13], [14] and [15], it is still far from being practical. We therefore only focus on additively homomorphic encryption in this thesis.

2.2.1 Operations in the encrypted domain

In this section we will focus on additively homomorphic cryptosystems, that is, cryptosystems for which Equation (2.1) holds and where the operations $*_1$ and $*_2$ can be replaced by an addition and a multiplication, respectively. This means that with the addition as operation, the set of plaintexts M forms an abelian group with neutral element 0 whereas the set of ciphertexts C forms a group with the multiplication as commutative operation and the element 1 as neutral element. Let $c_1, c_2 \in C$, then Equation (2.1) can be written as

$$D(c_1) + D(c_2) = D(c_1 \cdot c_2). \tag{2.2}$$

This implies that, given two elements $m_1, m_2 \in M$ and their encryptions $E(m_1)$ and $E(m_2)$, we can compute an encryption of $m_1 + m_2$ by calculating $E(m_1) \cdot E(m_2)$.

From now on we will treat the plaintexts m_1 and m_2 as two elements of some modular group $(\mathbb{Z}_n, +)$, where $n \in \mathbb{N}$. We will show how the homomorphic property enables us to perform certain arithmetic operations on them, even if only their encryptions $E(m_1)$ and $E(m_2)$ are known. In the remainder of this thesis, we will always assume that $m_1, m_2 \ll n$, i.e. that when performing arithmetic operations under the homomorphic encryption scheme overflows modulo n do not occur (except where explicitly stated).

2.2.1.1 Addition

The homomorphic property can be used to calculate the sum of two encrypted values: Given two encryptions $E(m_1)$ and $E(m_2)$, we can directly compute the encrypted sum $E(m_1+m_2)$ of the two plaintexts by multiplying their encryptions. This enables us to add two values m_1, m_2 available in encrypted form, without decrypting and encrypting the result. Note that this can also be done without knowing the private key.

2.2.1.2 Multiplication with constants

A second operation that can be performed on an encrypted value $E(m_1)$ is multiplication by a constant $k > 0$. That is, we can compute the encryption $E(k \cdot m_1)$ by raising $E(m_1)$ to the power of k. This is a direct result of applying Equation (2.2) $k-1$ times, as

$$E(m_1)^k = \underbrace{E(m_1) \cdot \ldots \cdot E(m_1)}_{k \text{ times}} = E(\underbrace{m_1 + \ldots + m_1}_{k \text{ times}}) = E(k \cdot m_1).$$

As the plaintext space is $\mathbb{Z}_n$, we can also work with negative numbers. For that we assume that the values $m_1, m_2 \in \mathbb{Z}_n$ are small enough (in comparison to the modulus n), so that reductions modulo n never occur. In this case we can interpret the values $n-1, n-2, \ldots$ as the negative numbers $-1, -2, \ldots$ In this representation we now can also multiply with integers $k < 0$. For such a k we raise the encryption $E(m_1)$ to the power of $n-k$ and accept that an overflow modulo n will occur:

$$\begin{aligned} E(m_1)^{(n-k)} &= E((n-k) \cdot m_1 \bmod n) \\ &= E(n \cdot m_1 - k \cdot m_1 \bmod n) \\ &= E(-k \cdot m_1). \end{aligned}$$

For completeness we mention that a multiplication with the constant value 0 can be achieved by computing an encryption of 0.

2.2.1.3 Subtraction

Given the ability to multiply with negative numbers, we can also subtract an encrypted value $E(m_2)$ from an encrypted value $E(m_1)$. This can easily be done by using the operations for addition and multiplication with the number -1 (which corresponds to the number $n-1 \in \mathbb{Z}_n$) in the homomorphic encryption scheme:

$$\begin{aligned} E(m_1 - m_2) &= E(m_1 + (-1) \cdot m_2) \\ &= E(m_1) \cdot E((-1) \cdot m_2) \\ &= E(m_1) \cdot E(m_2)^{-1}. \end{aligned}$$

2.2.2 The Paillier-Encryption Scheme

The Paillier cryptosystem [16] was presented by Pascal Paillier in 1999 and is one of the few practical homomorphic public-key cryptosystems. Its security is based on a computational problem called the *Composite Residuosity Class Problem*, namely the difficulty of deciding whether a number $m \in \mathbb{Z}_{n^2}^*$ is an n-th-residue modulo n^2. Paillier identified cryptosystems based on this computational problem as a new emerging class of cryptosystems, next to the ones that are based on the integer factorization problem such as *RSA* [17] and those that are based on the discrete logarithm problem such as *ElGamal* [18].

We will now describe the key generation process, the encryption and the decryption operation before we demonstrate that the cryptosystem has the homomorphic property.

2.2.2.1 Key Generation

In order to set up the cryptosystem, we first need to pick two random prime numbers p and q. We then set $n = p \cdot q$, and calculate the value of Carmichael's function $\lambda(n)$, in our case

$$\lambda := \lambda(n) = \operatorname{lcm}(p-1, q-1).$$

We then randomly choose an element $g \in \mathbb{Z}_{n^2}^*$ so that the order of g is a nonzero multiple of n. This can easily be validated by checking the equation

$$\gcd(L(g^\lambda \bmod n^2), n) = 1,$$

where L is the following function. Let $\mathcal{S}_n = \{u < n^2 \mid u = 1 \bmod n\}$, then $\forall u \in \mathcal{S}_n$ we define

$$L(u) = \frac{u-1}{n}.$$

As Paillier points out in [16], it usually suffices to choose $g = 2$ as a small base. We now consider the pair (n, g) as the public key, while the value λ serves as private key.

2.2.2.2 Encryption

As plaintext space we choose the numbers contained in $\mathbb{Z}_n$. In order to encrypt a message $m \in \mathbb{Z}_n$, we choose a random $r \in \mathbb{Z}_n$ and calculate the ciphertext

$$c = g^m \cdot r^n \bmod n^2.$$

Note that while we choose our message in $\mathbb{Z}_n$, we generate a ciphertext modulo n^2. Thus, the ciphertext space is $\mathbb{Z}_{n^2}^*$. We therefore accept a message expansion of (at least) a factor of two.

The cryptosystem offers probabilistic encryption, as for two different random values $r_1, r_2 \in \mathbb{Z}_n$ the message $m \in \mathbb{Z}_n$ will be encrypted to two different ciphertexts. Paillier proved in [16] that this encryption scheme satisfies the definition of semantic security.

Note that for a Paillier encryption $c = g^m \cdot r_1^n \bmod n^2$ of a plaintext m, we can obtain a new probabilistic encryption c' of m by choosing a random value r_2 and computing

$$\begin{aligned} c' &= c \cdot r_2^n \bmod n^2 \\ &= g^m \cdot r_1^n \cdot r_2^n \bmod n^2 \\ &= g^m \cdot (r_1 r_2)^n \bmod n^2. \end{aligned}$$

The process of computing a new random encryption, given a ciphertext, will be referred to as *re-encryption*.

Sometimes it is also useful to compute an encryption where the randomization part is chosen to be equal to 1, i.e. we compute

$$c = g^m \cdot 1 \bmod n^2.$$

This will be referred to as *deterministic encryption*, and is computationally much more lightweight than a full encryption. An example of a situation where deterministic encryptions are useful can be found in Section 5.1.2.2.

2.2.2.3 Decryption

To decrypt a ciphertext $c \in \mathbb{Z}_{n^2}^*$ we again use the function L and calculate:

$$m = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n.$$

For a proof of correctness we refer to Paillier's original paper [16].

2.2.2.4 Homomorphic property

In order to prove that the Paillier cryptosystem has the homomorphic property, we have to show that the Paillier decryption function $D_{sk} : C \to M$ is a homomorphism (of groups). That is, it is a map from the multiplicative group $(C, \cdot)$ to the abelian group $(M, +)$, where $C = \mathbb{Z}_{n^2}^*$ is the set of possible ciphertexts and $M = \mathbb{Z}_n$ is the set of plaintexts.

It can be seen that the function D_{sk} maps the neutral element 1_C of $\mathbb{Z}^*_{n^2}$ to the neutral element 0_M of $\mathbb{Z}_n$:

$$D_{sk}(1_C) = \frac{L(1^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n = \frac{0}{L(g^\lambda \bmod n^2)} \bmod n = 0_M.$$

Furthermore, given two ciphertexts $c_1, c_2 \in C$, we know that there exist $m_1, m_2 \in M$ and $r_1, r_2 \in \mathbb{Z}_n$ such that $c_1 = g^{m_1} {r_1}^n \bmod n^2$ and $c_2 = g^{m_2} {r_2}^n \bmod n^2$. Now we have

$$\begin{aligned}
D(c_1 \cdot c_2) &= D(g^{m_1} {r_1}^n \bmod n^2 \cdot g^{m_2} {r_2}^n \bmod n^2) \\
&= D(g^{m_1} g^{m_2} {r_1}^n {r_2}^n \bmod n^2) \\
&= D(g^{(m_1+m_2)} (r_1 r_2)^n \bmod n^2) \\
&= m_1 + m_2 \\
&= D(c_1) + D(c_2),
\end{aligned}$$

which shows that Equation (2.1) holds for the Paillier decryption function, since the product $r_1 r_2$ has the same distribution as r_1, r_2.

2.2.2.5 Optimizations and a Generalization

In [19], a generalization of the Paillier cryptosystem is presented, which is known as Damgård-Jurik cryptosystem. This generalization allows for a plaintext-space of size n^s, where n is the RSA modulus used in Paillier and $s \in \mathbb{N}$. Similar to Paillier, the encryption function is given by

$$c = g^m \cdot r^{n^s} \bmod n^{s+1}.$$

Thus, the Damgård-Jurik cryptosystem contains the encryption scheme by Paillier as the special case where $s = 1$.

Next, it is shown in [19] that the Paillier parameter g can always be set to $g = n+1$. For a given plaintext $m \in \mathbb{Z}_n$, this allows for a more efficient encryption. In particular for Paillier (the case $s = 1$), the encryption can be performed as

$$c = g^m \cdot r^n \bmod n^2 = (n+1)^m \cdot r^n \bmod n^2 = (1 + n \cdot m) \cdot r^n \bmod n^2,$$

thus one Paillier encryption requires only one exponentiation and two multiplications.

2.2.3 DGK Cryptosystem

In 2007, Damgård, Geisler and Krøigaard [20, 21] presented a homomorphic public-key cryptosystem which is semantically secure and has some interesting advantages compared to the Paillier cryptosystem. We will now give a short description of the DGK cryptosystem as it was proposed in [20, 21]. Compared to Paillier, the scheme has substantially smaller ciphertexts and the smaller plaintext- and ciphertext spaces result in a large performance gain.

2.2.3.1 Parameters

During the key generation process we choose public and private keys according to some parameters that determine the security and essential properties of the encryption scheme. Therefore, these parameters depend on the scenario in which the DGK encryption scheme will be used. The parameters k, t and ℓ defined here will be considered as public parameters that determine the structure of the cryptosystem.

The parameters k and t are security parameters: k determines the size of a RSA modulus and t, according to [20], should be large enough to make exhaustive search and other generic attacks infeasible. The parameter ℓ is the bitlength of values that need to be encrypted in the application scenario. If, for example, we want to design a comparison protocol where two 16-bit numbers are compared it suffices to set $\ell = 16$ [20]. As proposed in the original paper, realistic values for the given parameters are $k = 1024$, $t = 160$ and $\ell = 16$.

2.2.3.2 Key Generation

As in Paillier, we start with two primes p and q. These prime numbers should be generated in such a way that their product $n = pq$ forms a k-bit RSA modulus and that three additional prime numbers u, v_p, v_q, with the following properties exist: u is the smallest prime number greater than $\ell + 2$ and u is a divisor of both $p - 1$ and $q - 1$. The two values v_p, v_q should be t-bit prime numbers where v_p divides $p - 1$ and v_q divides $q - 1$.

In a second step we choose random values $g, h \in \mathbb{Z}_n^*$, so that the multiplicative order of h is $v_p v_q$, while we require g to be an element of order $u v_p v_q$ in $\mathbb{Z}_n^*$.

The plaintext space of the DGK cryptosystem is $M = \mathbb{Z}_u$. Note that, as the prime number u is quite small, also the plaintext space $\mathbb{Z}_u$ is small. Here lies one big advantage of the DGK cryptosystem: When designing protocols, we will many times raise an encrypted message to the power of a random element from the plaintext space (used for multiplicative blinding – see Section 3.2.2). As in the DGK cryptosystem the plaintext space is much smaller than for example in the Paillier cryptosystem, these exponentiations are more efficient as the exponents are quite small.

The ciphertext space is $C = \{g^m h^r \bmod n \mid m \in \mathbb{Z}_u, r \in \mathbb{Z}_{v_p v_q}\} \subseteq \mathbb{Z}_n^*$. We now set the public key to (n, g, h, u) and the private key to (p, q, v_p, v_q).

2.2.3.3 Encryption

In order to encrypt a message $m \in M$, we raise g to the power of m and multiply this with an uniformly random element of the subgroup generated by h. To obtain such an element we can choose a random element $r \in \mathbb{Z}_{v_p v_q}$ and compute h^r, as the order of the subgroup generated by h is $v_p v_q$. An encryption of m can thus be computed as

$$g^m h^r \bmod n.$$

Since both values v_p, v_q are part of the secret key, however, this can only be done by the owner of the secret key. Knowing only the public key, but not the value $v_p v_q$, one has to obtain the random element h^r in a different manner: We choose the value r as a random $2.5t$-bit integer. By choosing r in such a manner, we note that it is much larger than the value $v_p v_q$. For such a large random value r the value h^r will then be indistinguishable from a uniformly random element from the subgroup generated by h [20, 21]. In case we want to compute a deterministic encryption, we simply compute

$$g^m \bmod n,$$

i.e., we omit computation of the randomization part $h^r \bmod n$.

Similar to the Paillier cryptosystem, also the DGK encryption scheme is semantically secure (for a proof we refer to the original paper [20]). Therefore the cryptosystem also offers probabilistic encryption. Note that for a DGK encryption $c = g^m h^{r_1} \bmod n$ of a message m, we can re-encrypt the probabilistic encryption of m by choosing a random value r_2 and computing

$$\begin{aligned} c \cdot h^{r_2} \bmod n &= g^m h^{r_1} h^{r_2} \bmod n \\ &= g^m h^{r_1 + r_2} \bmod n. \end{aligned}$$

2.2.3.4 Decryption

Using the DGK cryptosystem, the owner of the secret key can effectively check whether or not a provided ciphertext c is an encryption of 0. This is due to the fact that

$$c^{v_p v_q} \bmod n = 1 \Leftrightarrow c \text{ encrypts } 0. \tag{2.3}$$

To decrypt an arbitrary ciphertext c, one computes the value $c^{v_p v_q} \bmod n$ and compares it to $(g^{v_p v_q})^m \bmod n$ for each $m \in M = \mathbb{Z}_u$. As the value u and therefore the number of possible values $(g^{v_p v_q})^m \bmod n$ is typically very small, one can easily store them in a look-up table for more efficient decryption. For a proof of correctness we refer to [20, 21].

2.2.3.5 Homomorphic property

The DGK cryptosystem is additively homomorphic. As for the Paillier cryptosystem, we will show that the decryption function $D : C \to M$ is a homomorphism. Equation (2.3) shows that $D(1) = 0$, so it remains to see that Equation (2.1) holds. For two ciphertexts $c_1, c_2 \in C$, we know that there exist $m_1, m_2 \in M$ and $r_1, r_2 \in \mathbb{Z}_{v_p v_q}$ with $c_1 = g^{m_1} h^{r_1} \bmod n$ and $c_2 = g^{m_2} h^{r_2} \bmod n$. Then it holds

$$\begin{aligned} D(c_1 \cdot c_2) &= D((g^{m_1} h^{r_1} \bmod n) \cdot (g^{m_2} h^{r_2} \bmod n)) \\ &= D(g^{m_1} g^{m_2} h^{r_1} h^{r_2} \bmod n) \\ &= D(g^{(m_1+m_2)} h^{(r_1+r_2)} \bmod n) \\ &= m_1 + m_2 \\ &= D(c_1) + D(c_2). \end{aligned}$$

In the next sections we describe other important cryptographic primitives which will be used in the remainder of this thesis.

2.3 Oblivious Pseudo-Random Function Evaluation

An Oblivious Pseudo-Random Function (OPRF) is a two-party protocol, where one party holds a secret key k and a second party wishes to evaluate a keyed pseudo-random function $F_{\text{PRF}}(k, x)$ on a value x. The function evaluation is oblivious in the sense that the party holding the key k does not learn x, while the other party only obtains $F_{\text{PRF}}(k, x)$ and has no knowledge of k.

The first construction of an OPRF was presented in [22] and has later been modified in [23]. Important applications of OPRFs are secure protocols for keyword search [22], set intersection [23] and adaptive oblivious transfer [24].

In this thesis we build secure protocols using the OPRF presented in [24] (independently and concurrently also in [25]). Their protocol securely realizes the function

$$F_{\mathrm{PRF}}(k,x) = \mathfrak{g}^{\frac{1}{k+x}},$$

where $\mathfrak{g}$ is the generator of a multiplicative group of order n, where n is an RSA modulus. Our variation of their OPRF will be presented in Section 5.1.1. For details on their construction, security assumptions and security proofs we refer to [24].

2.4 Oblivious Transfer

Oblivious Transfer (OT) is an important cryptographic primitive which allows two parties to obliviously transfer one value out of a set of n values. In particular, party $\mathcal{A}$ has access to a list of values $x_1, \ldots, x_n$, while party $\mathcal{B}$ requests the value present at the i-th position (with $1 \leq i \leq n$). OT allows to perform this operation securely so that $\mathcal{B}$ *only* obtains the requested x_i but not the other values, while $\mathcal{A}$ does not see which value was requested (i.e., the index i).

In this thesis we will make use of an efficient protocol for OT that can be found in [26]. Their solution consists of an initialization and an online phase. The overhead of the initialization phase can be amortized over all consecutive OT executions and is therefore negligible for most applications. The complexity of the online phase mainly consists of three exponentiations. For details we refer to [26].

2.5 Notation

We introduce some notation that will be used in the remainder of this thesis. When we randomly and uniformly pick a value r from a (finite) set of numbers S, we shortly write

$$r \in_R S.$$

When we use square brackets (e.g. $\langle\cdot\rangle, [\cdot]$ or $[\![\cdot]\!]$), we refer to a homomorphic encryption of the value inside the brackets. By writing $[m]$ for some $m \in \mathbb{Z}_n$ we denote a Paillier encryption $g^m \cdot r^n \bmod n^2$ of m, where $r \in_R \mathbb{Z}_n$. Similarly, we use $[\![m]\!]$ for an encryption in the DGK cryptosystem. To denote that any homomorphic encryption scheme can be used, we write $\langle m\rangle$.

To denote that a value m is to be encrypted, we write

$$m \Rightarrow [m] \text{ or } E(m).$$

To denote decryption, we write

$$m \Leftarrow [m] \text{ or } D([m]).$$

Note that, regardless of the cryptosystem in use, we will omit to explicitly denote the public or private key which is used to perform the cryptographic operations. From the context it will always be clear, which public or private key should be used. A re-randomization of an encrypted value $[m]$ will be denoted by $[m]_{re\text{-}rand}$.

Chapter 3

Secure Computations on Integer Values

In this chapter we describe the three major approaches to secure multiparty computation, namely garbled circuits, homomorphic encryption and secret sharing. Each of these approaches has its own strengths and weaknesses, and therefore deserves special attention. The secure protocols described in the remainder of this thesis are constructed mainly using homomorphic encryption and garbled circuits. The main advantage of garbled circuits lies in its efficiency when performing non-linear operations, such as algorithms requiring the bit representation of a given value. However, as we will see, garbled circuits have a large expansion factor when encrypting inputs, e.g. one ciphertext typically consumes more than 80 times the space of its plaintext representation. This yields to secure protocols with a high communication overhead. The approach of SMC via homomorphic encryption is more efficient in this sense, but has drawbacks when it comes to simple, but non-linear operations such as a comparison. For this reason in Chapters 5 and 6 we will use a combination of these techniques to build our secure protocols.

All secure protocols presented in this chapter and the remainder of this thesis are secure in the *honest-but-curious* model of secure computation (see Section 2.1), i.e., under the assumption that both participants faithfully follow the protocol specification. While the parties have to follow the protocols, they are allowed to record all communication and draw conclusions from the observations they make.

3.1 Garbled Circuits

Garbled Circuits (GC) have been introduced in [27] and are a general solution for SMC. Since the work of [27], garbled circuits have been frequently improved. In this thesis, we use the construction proposed in [28] with optimizations in [29, 30]. This allows XOR-gates to be evaluated at essentially no cost and each garbled circuit table consists of three entries, instead of four. When evaluating the complexity of our protocols, we will only count the gates other than XORs. In this section we give a brief description of the basic GC techniques.

W_1	W_2	$W_3 = W_1 \wedge W_2$
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: Truth table for an AND gate with input wires W_1, W_2.

In the basic version of garbled circuits, two parties $\mathcal{A}$ and $\mathcal{B}$ want to privately evaluate a function $f(a, b)$ on their respective inputs a and b. The function f needs to be specified as a Boolean circuit. In order to preserve the privacy of both parties, the circuit needs to be *garbled*. For this, party $\mathcal{B}$ chooses two random keys K_0^W, K_1^W for each input wire W of the circuit. For each gate he writes down the truth table, e.g. see Table 3.1 for an AND gate with input wires W_1, W_2 and output wire W_3.

W_1	W_2	K^{W_1}	K^{W_2}	$E(K^{W_3})$
0	0	$K_0^{W_1}$	$K_0^{W_2}$	$E_{K_0^{W_1} \| K_0^{W_2}}(K_0^{W_3})$
0	1	$K_0^{W_1}$	$K_1^{W_2}$	$E_{K_0^{W_1} \| K_1^{W_2}}(K_0^{W_3})$
1	0	$K_1^{W_1}$	$K_0^{W_2}$	$E_{K_1^{W_1} \| K_0^{W_2}}(K_0^{W_3})$
1	1	$K_1^{W_1}$	$K_1^{W_2}$	$E_{K_1^{W_1} \| K_1^{W_2}}(K_1^{W_3})$

Table 3.2: Encrypted truth table for an AND gate.

Next, for each output wire he encrypts the corresponding key with the two keys assigned to the input values (Table 3.2). Finally, $\mathcal{B}$ randomly permutes the encrypted values to obtain a *garbled gate* (Table 3.3). These garbled gates will be sent to party $\mathcal{A}$.

In order to decrypt the garbled table entries, $\mathcal{A}$ needs the keys of the input

Permutation of $E(K^{W_3})$
$E_{K_0^{W_1}\|\|K_1^{W_2}}(K_0^{W_3})$
$E_{K_0^{W_1}\|\|K_0^{W_2}}(K_0^{W_3})$
$E_{K_1^{W_1}\|\|K_1^{W_2}}(K_1^{W_3})$
$E_{K_1^{W_1}\|\|K_0^{W_2}}(K_0^{W_3})$

Table 3.3: Garbled AND gate.

wires. The keys for the inputs of $\mathcal{B}$ can be sent directly to $\mathcal{A}$, along with the circuit. $\mathcal{A}$ can obtain the keys for her inputs by using oblivious transfer (see Section 2.4), in a way that she receives exactly one of the two keys K_0^W, K_1^W for her input wires, and stays oblivious about the other. $\mathcal{A}$ can now iteratively decrypt each garbled gate, obtaining the encryption keys of the next gate. The GC protocol terminates depending on who should receive the final output of the computation: In case that party $\mathcal{A}$ is supposed to learn the output, $\mathcal{B}$ creates the output gates differently while generating the circuit. In this case, the output gates directly reveal the final output of this respective output gate, e.g. either 0 or 1. In case that $\mathcal{B}$ is supposed to receive the output, $\mathcal{A}$ sends the final output keys to $\mathcal{B}$.

Rather than choosing the keys for all wires randomly, it suffices for $\mathcal{B}$ to choose only the keys K_0^W for non-XOR gates at random. The corresponding keys K_1^W will then be defined as $K_1^W := K_0^W \oplus S$, for some fixed random string S. This way, an XOR gate of two given keys $K_i^{W_1}, K_j^{W_2}$ with $i, j \in \{0, 1, \}$ can be computed as $K_i^{W_1} \oplus K_j^{W_2}$. Thus, no garbled truth table is needed.

Furthermore, for any non-XOR gate, we can specify the key $K_0^{W_3}$ to be an encryption of $K_0^{W_1}, K_0^{W_2}$. This allows this value to be computed directly by $\mathcal{A}$, thus it does not need to be included in the garbled truth table.

For simplicity of notation, we will use Boolean operators to denote binary gates, e.g. an AND gate with inputs a and b will be denoted by ab, OR gates by $a + b$ and XOR by $a \oplus b$. The negation $\neg$ will be realized as an XOR gate with one constant input 1, a multiplexer will be denoted by $\text{MUX}_a(b, c)$, where $\text{MUX}_a(b, c) = b$ if $a = 1$ and else outputs c. We use K_a to denote a garbled circuit input corresponding to the boolean value a. Evaluation of Boolean operators as a garbled circuit will be denoted by $\{\cdot\}_{GC}$, e.g. we write $\{a + b\}_{GC}$ to denote evaluation of a garbled OR gate on inputs a and b.

3.2 Secret Sharing

A second approach to realize SMC is the use of *secret sharing.* In order to see how a SMC protocol can be designed using secret sharing [31], we first describe how secret sharing works. Here we will only shortly review some specifics of secret sharing schemes, for a more detailed introduction into SMC and secret sharing we refer to [32].

In an n-party secret sharing scheme, a secret value s can be broken up into secret shares $s_1, \ldots, s_n$ in a way that a subset of all these shares is necessary to reconstruct the value s. To give concrete examples, in the following we will focus on a two-party scenario, where the parties $\mathcal{A}$ and party $\mathcal{B}$ each share a value. However, all results can easily be generalized to the n-party setting.

One simple example for a secret sharing scheme would be using addition in a finite ring or additive group: Each of the two parties additively breaks her input into shares (in a way that both shares are uniformly random elements) and sends one share to the other party. For example $a = a_1 + a_2$ and $b = b_1 + b_2$. It is obvious that each value can be reconstructed if and only if both shares are known.

Let us now assume that each party has shared her private input with the other party. That is, party A knows the share b_1 of the input b and party B knows the input a_2 of the value a. Let f be a function that should be evaluated in a secure way on the inputs a and b from our two parties, so the desired output will be $f(a, b)$. We call a secret sharing scheme *homomorphic* under the function f, if each party can compute a share of the result of the function using her shares. Finally the result $f(a, b)$ can be reconstructed from these shares. In our example, if the additive secret sharing scheme is homomorphic under the function f, party A computes $y_A = f(a_1, b_1)$ whereas party B computes $y_B = f(a_2, b_2)$. Then the value $f(a, b)$ can be reconstructed from the shares y_A and y_B. It is easy to see that the above additive secret sharing scheme is homomorphic under addition.

The probably most famous secret sharing scheme was proposed by Adi Shamir in 1979 [31] and it can be used for general SMC [32]. Concerning the security model, realizing SMC using a secret sharing scheme such as the Shamir scheme typically leads to SMC techniques with *perfect security* (see [10], Section 7.6.1). This is due to the fact that it is impossible for an attacker to learn anything from the shares he receives, no matter how computational powerful he is. If a SMC protocol based on secret sharing should be made secure against active attackers, however, more advanced techniques such as verifiable secret sharing are necessary [33].

In this thesis, two weak instances of secret sharing will be used, and will be referred to as *blinding*. These basic techniques will be used to obfuscate certain values while still being able to work with them (under the implied homomorphism). There are two basic approaches for blinding: Additive and multiplicative blinding. We will present a short overview of these blinding techniques in this section. We assume that all computations are done in a modular group $\mathbb{Z}_n$. Let X be a random variable over $\mathbb{Z}_n$, and $P(X = x)$ denotes the probability that X takes on the value x. For a fixed value $x \in \mathbb{Z}_n$ we will now see how additive and multiplicative blinding can be used to obfuscate the value x.

3.2.1 Additive Blinding

There are two ways to additively blind the value x. The first method produces a uniformly random element $y \in \mathbb{Z}_n$ such that y does not reveal any information about the value x nor the distribution of X. First we choose a random element $r \in \mathbb{Z}_n$, where the distribution of r is the discrete uniform distribution. This means, we sample from a random variable R which takes on values $r \in \mathbb{Z}_n$ with the same probability $P(R = r) = \frac{1}{n}$ for all $r \in \mathbb{Z}_n$. We then compute $y = r + x \bmod n$. It can easily be seen that the element y is a uniformly random element of the set $\mathbb{Z}_n$, and therefore reveals no information about the value x.

Sometimes it is desired to not produce an overflow modulo n when adding the random element r. This is possible, if the statistical distance κ between the distributions of X and R is large enough. In this case, the probability of revealing information about the value x and the distribution of X is negligible in the security parameter κ. For further details we refer to Appendix A of [34].

3.2.2 Multiplicative Blinding

We will now assume that the random variable X takes on values from a field $\mathbb{Z}_p$, where p is a prime number. When using *multiplicative blinding*, we usually want to obfuscate the value of x in case that $x \neq 0$. For this reason, when using multiplicative blinding, the value $x \in \mathbb{Z}_p$ is multiplied by a random value $r \in \mathbb{Z}_p \backslash \{0\}$. Then the result $y = x \cdot r \bmod p$ is equal to 0 if and only if $x = 0$. In case $x \neq 0$, the value y should not reveal any information about the original value x, that is, y should be a uniformly random element of $\mathbb{Z}_p \backslash \{0\}$. This indeed is the case, as the following theorem shows:

Theorem. *Let R be a random variable with uniform distribution on $\mathbb{Z}_p \backslash \{0\}$.*

For any $x \in \mathbb{Z}_p \backslash \{0\}$ the random variable $Y = x \cdot R \bmod p$ is again uniform on $\mathbb{Z}_p \backslash \{0\}$.

Proof. As the random variable R takes values on $\Omega_R = \{1, \ldots, p-1\}$, the random variable $Y = x \cdot R$ only takes values on $\Omega_Y = \{x \cdot 1, \ldots, x \cdot p - 1\}$. As in the field $\mathbb{Z}_p$ there are no zero divisors and $x \neq 0$, the map $f : \mathbb{Z}_p \to \mathbb{Z}_p$, $a \mapsto x \cdot a$ is a bijective homomorphism. Therefore Ω_R and Ω_Y are of the same cardinality $p - 1$ and we see that $\Omega_R = \Omega_Y$. Now $P(Y = x \cdot r) = P(R = r)$ for all $r \in \Omega_R$, in other words Y is uniform on Ω_Y. $\square$

Note that, when working with values from a plaintext space of a public key cryptosystem (such as $\mathbb{Z}_n$ or $\mathbb{Z}_{n^2}$ with n being a composite), the theorem given here is in general not correct. This is due to the fact that the plaintext space of such a cryptosystem is a ring rather than a field. In this case there are pairs x, r for which $P(Y = x \cdot r) \neq P(R = r)$. However, the probability of randomly selecting such a pair is negligible.

3.3 Homomorphic Encryption

A third way to build SMC protocols is to make use of a homomorphic public-key cryptosystem. In the two party setting, it suffices to use a standard homomorphic cryptosystem where one of the parties knows the public key. In case that more than two parties participate in the computations, a threshold cryptosystem should be used (see Section 2.2). In the latter case, encryptions can be computed by every party but ciphertexts can only be decrypted when certain parties cooperate. Since in this thesis we will work in the two party setting, we will only give a short description of the case for $n \geq 3$ parties, and then focus on the two party setting.

Initially, in order to compute the result of an arithmetic function, the involved parties set up a homomorphic cryptosystem. Every party can encrypt her input using the public key. She then can provide this encrypted value to all the participating parties, as under the threshold scheme they cannot decrypt ciphertexts without the cooperation of the remaining parties. Using the homomorphic property, as seen in Section 2.2.1, every party can then add or subtract two encrypted values, or multiply them with arbitrary constants that are known as clear texts.

In Section 2.2.1 we have already seen how we can perform arithmetic operations such as additions and multiplications with known constants in homomorphic encryption schemes on the encrypted values even without actually knowing the values themselves or how to decrypt them. If we could use a fully homomorphic cryptosystem (see Section 2.2) one could already compute any given function under the homomorphism since the fully homomorphic scheme allows for both additions and multiplications. However, since efficient fully homomorphic cryptosystems are not available (and it is not clear whether they will be in the future), we will see how to realize a secure multiplication using an additive homomorphic cryptosystem. This allows to construct a general approach for SMC using homomorphic ciphers.

3.3.1 Secure Multiplication

While additions and multiplications with constants are immediately possible when using homomorphic encryption, multiplications of ciphertexts require execution of a cryptographic protocol such as the one that was presented by Cramer, Damgård and Nielsen [35]. Here we only give a short sketch of the protocol; for a more detailed version and a proof of security we refer to the original paper [35]. In an n-party setting, we start with two encrypted values $\langle a\rangle$ and $\langle b\rangle$, and we want to compute the encrypted product $\langle c\rangle = \langle a \cdot b\rangle$. This can be done in 5 steps according to Protocol 1.

Protocol 1 Secure multiplication for $n > 2$ parties

Input: $\langle a\rangle, \langle b\rangle$

Output: $\langle ab\rangle$

```
1: Each party $P_i$ chooses a random value $r_i$ and broadcasts an encryption of it.
2: All parties now can compute $\langle r\rangle = \langle r_1 + \ldots + r_n\rangle$ and, using the homomorphic property, an encryption of $a+r$. Then $\langle a+r\rangle$ is decrypted jointly using threshold decryption, so that every party learns the value $a+r$.
3: Now the first party sets $a_1 = (a+r) - r_1$ and the remaining parties set $a_i = -r_i$. Observe that $a = a_1 + \ldots + a_n$.
4: Each party now computes $\langle a_i \cdot b\rangle = \langle b\rangle^{a_i}$ and broadcasts it.
5: Now each party computes
       $$\langle a_1 \cdot b\rangle \cdot \ldots \cdot \langle a_n \cdot b\rangle = \langle (a_1 + \ldots + a_n) \cdot b\rangle = \langle a \cdot b\rangle.$$
```

Secure Multiplication in the two-party case. In this thesis, we work in the following setting: Party $\mathcal{A}$ holds a private key for some homomorphic encryption scheme. Party $\mathcal{B}$ holds two encrypted values $\langle a\rangle, \langle b\rangle$ and should obtain an encryption $\langle ab\rangle$. This can be achieved as depicted in Protocol 2. First, party $\mathcal{B}$ generates two random values r_a, r_b and computes encryptions $\langle a + r_a\rangle$ and $\langle b + r_b\rangle$. $\mathcal{B}$ sends these values to $\mathcal{A}$, who subsequently decrypts, multiplies them and sends an encryption of $(a+r_a)(b+r_b)$ back to party $\mathcal{B}$. Party $\mathcal{B}$ now recovers the product by computing

$$\langle ab\rangle = \langle (a + r_a)(b + r_b)\rangle \langle b\rangle^{-r_a} \langle a\rangle^{-r_b} \langle -r_a r_b\rangle.$$

If values a and b are known to be from the set $\{0, 1\}$, the operations become much more lightweight. In this case the blinding factors r_a, r_b can also be chosen as binary values. Party $\mathcal{A}$ now sends $\langle a \oplus r_a\rangle$ and $\langle b \oplus r_b\rangle$ to party $\mathcal{B}$, who subsequently sends back the product $\langle (a \oplus r_a)(b \oplus r_b)\rangle$. Here, we use $\oplus$ to denote a bitwise XOR which can be computed on homomorphic encryptions $\langle x\rangle$ and $\langle y\rangle$ as

$$\langle x \oplus y\rangle = \langle x\rangle \langle y\rangle \langle x\rangle^{-2y} \text{ for } x, y \in \{0, 1\}.$$

As before, using the values r_a, r_b party $\mathcal{B}$ recovers the result $\langle ab\rangle$. We denote a secure multiplication by $*$, a secure multiplication of binary values by $\circledast$. Note that both protocols are only secure in the honest but curious attacker model. In particular, the protocol for binary multiplication is only secure if the values a, b are binary.

3.3.2 Integer Comparison

An important primitive in SMC is the comparison of two integer values. A basic version of this problem is known as the millionaires' problem [1]. In order to use it in larger SMC protocols we require a variant of this problem which allows to also compare two *encrypted* values (e.g. see Section 5.2.4.3 for an example). For this reason we describe a protocol which allows to compare two encrypted ℓ-bit values $\langle a\rangle$, $\langle b\rangle$ and consequently allows to select the minimum along with some encrypted value, associated to the minimum. In particular given two tuples $(\langle a\rangle, \langle V_a\rangle)$ and $(\langle b\rangle, \langle V_b\rangle)$, the protocol computes $(\langle c\rangle, \langle V_c\rangle)$ where $c = \min(a, b)$ and $V_c = V_{\min(a,b)}$. This problem differs from the standard millionaires' problem, since both input values *and* the output need to remain hidden from both parties. A solution to this problem has been presented in [36, 37] and is summarized here:

Protocol 2 Secure multiplication in the two-party setting.

Input: $\langle a \rangle, \langle b \rangle$

Output: $\langle ab \rangle$

```
1: Party $\mathcal{B}$:
   Choose random values $r_a, r_b$.
   Compute $\langle a' \rangle = \langle a + r_a \rangle$ and $\langle b' \rangle = \langle b + r_b \rangle$.
   Send $\langle a' \rangle, \langle b' \rangle$ to party $\mathcal{A}$.
2: Party $\mathcal{A}$:
   $a' \Leftarrow \langle a' \rangle, b' \Leftarrow \langle b' \rangle$ // Decrpyt
   $a'b' \Rightarrow \langle a'b' \rangle$ // Encrpyt
   Send $\langle a'b' \rangle$ to party $\mathcal{B}$
3: Party $\mathcal{B}$:
   Compute $\langle ab \rangle = \langle a'b' \rangle \langle b \rangle^{-r_a} \langle a \rangle^{-r_b} \langle -r_a r_b \rangle$
```

Initially party $\mathcal{B}$, who has access to both $\langle a \rangle$ and $\langle b \rangle$, computes

$$\langle z \rangle = \langle 2^\ell + a - b \rangle = \langle 2^\ell \rangle \cdot \langle a \rangle \cdot \langle b \rangle^{-1},$$

where ℓ denotes the maximal bit length of a and b. As $0 \leq a, b < 2^\ell$, z is a positive $(\ell + 1)$-bit value. Moreover, it holds for z_ℓ, the most significant bit of z, that

$$z_\ell = 0 \Leftrightarrow a < b.$$

Given an encryption of $z \bmod 2^\ell$, the result can be computed immediately:

$$z_\ell = 2^{-\ell} \cdot (z - (z \bmod 2^\ell)).$$

Once $\mathcal{B}$ has an encryption of the outcome $\langle z_\ell \rangle = \langle a < b \rangle$, an encryption of the minimum m, is easily obtained using arithmetic, as $m = (a < b) \cdot (a - b) + b$. Determining an encryption of the value V_c is analogous, $(a < b) \cdot (V_a - V_b) + V_b$. It remains to describe how $\mathcal{B}$ obtains the encryption of $z \bmod 2^\ell$.

First, party $\mathcal{B}$ generates a uniformly random $(\kappa + \ell + 1)$-bit value r, where κ is a security parameter, say 100, and $\kappa + \ell + 1 \ll \log_2(n)$. This will be used to additively blind z,

$$\langle d \rangle = \langle z + r \rangle = \langle z \rangle \cdot \langle r \rangle;$$

$\langle d \rangle$ is then re-randomized and sent to $\mathcal{A}$ who decrypts it and reduces d modulo 2^ℓ. The obtained value is then encrypted, and returned to $\mathcal{B}$.

Due to the restriction on the bit-length of r, party $\mathcal{B}$ can now *almost* compute the desired encryption $\langle z \bmod 2^\ell \rangle$. The masking can be viewed as occurring over the integers, thus we have $d \equiv z + r \bmod 2^\ell$ and

$$\left(z \bmod 2^\ell\right) = \left(\left(d \bmod 2^\ell\right) - \left(r \bmod 2^\ell\right)\right) \bmod 2^\ell.$$

$\mathcal{A}$ has just provided $\langle d \bmod 2^\ell \rangle$ and r is known to $\mathcal{B}$. Thus, he can compute

$$\langle \tilde{z} \rangle = \langle (d \bmod 2^\ell) - (r \bmod 2^\ell) \rangle = \langle d \bmod 2^\ell \rangle \cdot \langle (r \bmod 2^\ell) \rangle^{-1}.$$

Had the secure subtraction occurred modulo 2^ℓ, $\tilde{z}$ would be the right result; however, it occurs modulo n. Note, though, that if $d \bmod 2^\ell \geq r \bmod 2^\ell$, $\tilde{z}$ is the right result. On the other hand, if $r \bmod 2^\ell$ is larger, an underflow has occurred; adding 2^ℓ in this case gives the right result. So, if $\mathcal{B}$ had an encryption $\langle \lambda \rangle$ of a binary value indicating whether $r \bmod 2^\ell > d \bmod 2^\ell$, he could simply compute

$$\langle z \bmod 2^\ell \rangle = \langle \tilde{z} + \lambda 2^\ell \rangle = \langle \tilde{z} \rangle \cdot \langle \lambda \rangle^{2^\ell},$$

which adds 2^ℓ exactly when $r \bmod 2^\ell$ is the larger value. This leaves us with a variant of Yao's millionaires problem: $\mathcal{B}$ must obtain an encryption $\langle \lambda \rangle$ of a binary value containing the result of the comparison of two private inputs: $\hat{d} = d \bmod 2^\ell$ held by $\mathcal{A}$ and $\hat{r} = r \bmod 2^\ell$ held by $\mathcal{B}$.

The solution to this final problem is based on Damgård et al. [20, 21]. In their work, for efficiency reasons they use the DGK cryptosystem which was simultaneously proposed in [20, 21] (see Section 2.2.3 for details). Though the basic setting of Damgård et al. considers one public and one secret value, they note how to construct a solution for private inputs. They also note how to obtain a secret output. However, they obtain this output as an additive secret sharing, while in our setting $\mathcal{B}$ must receive a homomorphic encryption (either DGK or Paillier) $\langle \lambda \rangle$ at the end of the protocol. Naturally $\mathcal{A}$ must not see this encryption as she knows the secret key.

We assume that $\mathcal{A}$ has run the DGK key-generation algorithm and has sent the public key to $\mathcal{B}$. This key pair can be re-used whenever the comparison protocol will be run. Initially, $\mathcal{A}$ sends $\mathcal{B}$ encryptions of the bits of her input, $[\![\hat{d}_{\ell-1}]\!], \ldots, [\![\hat{d}_0]\!]$. $\mathcal{B}$ then chooses $s \in_R \{1, -1\}$ and computes

$$[\![c_i]\!] = [\![\hat{d}_i - \hat{r}_i + s + 3 \sum_{j=i+1}^{\ell-1} w_j]\!] = [\![\hat{d}_i]\!] \cdot [\![-\hat{r}_i]\!] \cdot [\![s]\!] \cdot \left(\prod_{j=i+1}^{\ell-1} [\![w_j]\!] \right)^3, \qquad (3.1)$$

where $[\![w_j]\!] = [\![\hat{d}_j \oplus \hat{r}_j]\!]$, which he can compute as $\mathcal{B}$ knows $\hat{r}_j$. For technical reasons (to avoid the case $\hat{d} = \hat{r}$), we append differing bits to both $\hat{d}$ and $\hat{r}$, i.e., we compare the values $2\hat{d} + 1$ and $2\hat{r}$ instead.

Equation (3.1) differs from the one proposed by Damgård et al. in order to efficiently hide the output, but the core idea remains. Consider the case of $s = 1$; if $\hat{d}$ is larger, then all c_i will be non-zero. However, if $\hat{r}$ is larger, then exactly one c_i will equal zero, the one at the most significant differing bit-position. Both claims are easily verified. For $s = -1$ we have exactly the same situation, except that the zero occurs if $\hat{d}$ is larger. The factor of 3 ensures that the values are non-zero once even a single w_j is set.

$\mathcal{B}$ now multiplicatively masks the $[\![c_i]\!]$ with a uniformly random $r_i \in \mathbb{Z}_u^*$

$$[\![e_i]\!] = [\![c_i \cdot r_i]\!] = [\![c_i]\!]^{r_i},$$

re-randomizes and permutes the encryptions $[\![e_i]\!]$ and sends them to $\mathcal{A}$. Note that e_i is uniformly random in $\mathbb{Z}_u^*$ except when $c_i = 0$, in which case e_i also equals zero, i.e. the existence of a zero is preserved.

$\mathcal{A}$ now decrypts all e_i and checks whether one of them is zero. She then encrypts a bit $\tilde{\lambda}$, stating if this is the case. At this point she switches back to the homomorphic cryptosystem denoted by $\langle\cdot\rangle$, i.e. $\mathcal{A}$ sends $\langle\tilde{\lambda}\rangle$ to $\mathcal{B}$. Given the knowledge of s, $\mathcal{B}$ can compute the desired encryption $\langle\lambda\rangle$: while $\langle\tilde{\lambda}\rangle$ only states whether there was a zero among the values decrypted by $\mathcal{A}$, s explains how to interpret the result, i.e. whether the occurrence of a zero means that $\hat{r} > \hat{d}$ or $\hat{d} \geq \hat{r}$. In the former case, $\mathcal{B}$ negates the result $\langle\tilde{\lambda}\rangle$ under encryption by computing $\langle\lambda\rangle = \langle 1 - \tilde{\lambda}\rangle$. Otherwise he directly takes $\langle\tilde{\lambda}\rangle$ as output $\langle\lambda\rangle$. For more details and a graphical presentation we refer to [36].

3.4 Related Work

We use this section to summarize and cite related work which is not covered in detail by Chapters 2 and 3, but should be mentioned for completeness reasons.

3.4.1 SMC on Binary and Integer Arithmetic

Several constructions for Secure Multiparty Computation are known in the literature. Yao's famous Millionaires problem [1] and the first general approach to SMC [2] are mentioned in the introduction of this thesis.

Various other approaches for securely evaluating a function have been developed for different function representations, namely combinatorial circuits [27, 38], Ordered Binary Decision Diagrams [39], branching programs[40], homomorphic threshold encryption [35] or one-dimensional look-up tables [41]. While the latter solve the problem of secure function evaluation in the semi-honest attacker scenario, the works of [42, 43] present a solution for the malicious case.

A number of frameworks have been proposed which are supposed to hide cryptographic details from the user and allow for a direct implementation of privacy-preserving applications: Fairplay [44], VIFF [45] and TASTY [46] constitute important examples. Parts of our protocols are ready to be implemented in either of these frameworks.

3.4.2 SMC on Non-Integer Values

While the methods presented above target computations performed over the integers only, extensions were proposed that can handle rational numbers [47] and real values in fixed point notation [48, 49]. The proposed framework [48] presents secure protocols for all arithmetic operations in a multiparty setting with 3 or more parties. All computations are performed on scaled and quantized values with fixed precision. While all arithmetic operations are rather efficient, the proposed framework comes with a major drawback: In order to prevent scaling factors to accumulate, after each arithmetic operation a truncation step needs to be performed, which may introduce an error in the least significant bit of the result. This is an effect of a probabilistic rounding operation. The work presented in [49] proposes an efficient technique which overcomes this rounding problem in the two-party setting. Their work can easily be extended to a framework for fixed point computations (see Chapter 4), with better performance than [48]. Even though very efficient, however, all these approaches do not provide numerical stability when performing a large number of arithmetic operations on very small values; indeed there are applications where it is known that a fixed point representation does not yield to accurate results. For example, we will see in Chapter 7 and 8 that a log encoding based representation is advantageous for the considered application scenario, when compared to a fixed point implementation with sufficiently large precision, both in terms of accuracy and efficiency.

3.4.3 Selected SMC Applications

There has been an increasing interest in the use of Secure Multiparty Computation to enhance privacy in auctions [50], data clustering [51, 52], analysis of medical signals [53], to name only a few application scenarios.

In particular, the field of signal processing in the encrypted domain has become an important field of research, and can benefit from the techniques presented in this dissertation. Processing of encrypted signals aides to enhance privacy in many sensitive applications, such as classification of signals in medical applications [54] or evaluation of biometrics on encrypted data [37]. Typically, these applications come from domains that require processing of real-valued signals.

Chapter 4

Secure Computations in Fixed Point Representation

Several strategies to securely compute with non-integer values have been discussed in the literature. The usual way to deal with this problem is to quantize, i.e. to compute with scaled and rounded approximations of real values. In fact, this strategy corresponds to the implementation of a fixed point representation. Even though the techniques presented in this chapter can be considered as folklore they will be detailed here to serve as a basic reference implementation in Chapter 7.

The ability to perform secure computations using a fixed point representation already allows for a large class of applications to be implemented. However, there are still many algorithms which can not be realized by these techniques. Examples are algorithms for (non-) linear or dynamic programming which typically work on very small probabilities, or numerical computations such as solving linear equations, singular value decomposition etc. As we will see in Chapter 7, these problems would typically require fixed point arithmetic with a precision that lies out of a practical scope.

4.1 General Idea

Given the methods described in Chapter 3 to compute on integer values, it is straightforward to describe a computational framework which allows to compute on non-integer values in a fixed point representation. Assume we want to perform computations on values with a fractional part of at most ℓ bits. Then it suffices to multiply each value with a scale factor of 2^ℓ and round to the nearest integer. Now, all operations can be performed using integer arithmetic. After each multiplication, the scale factor accumulates. Care must be taken if two values should be added which have different scale factors; in this case the value with the smaller scale factor needs to be adjusted. Given that most schemes for SMC only allow for a limited plaintext space, the above described scheme only allows for a very small number of consecutive multiplications.

4.2 Rounding Strategies

In order to overcome this restriction, protocols have been presented which allow to rescale the result of an multiplication [48, 49]. There are two kinds of protocols which achieve this functionality, so called probabilistic and deterministic rounding protocols.

4.2.1 Probabilistic Rounding Protocol

The highly efficient probabilistic protocols in [48] and in [49] randomly introduce an 1-bit error in the least significant bit. With time, these random errors can accumulate and propagate to the higher order bits and therefore invalidate the results of the computations. For this reason, this drawback makes the protocol only applicable for small applications that do not require too many consecutive multiplications.

Protocol 3 Probabilistic Rescaling Protocol

Input: $(\mathcal{B})$ $\langle a_{2\ell} \rangle$ with 2ℓ-bit quantization

Output: $(\mathcal{B})$ $\langle a_{\ell} \rangle$ with ℓ-bit quantization

```
1: Party $\mathcal{B}$:
   Additively blind the value $\langle a_{2\ell} \rangle$ with some value $-r$, chosen uniformly at random.
   Send $\langle a_{2\ell} - r \rangle$ to party $\mathcal{A}$.
2: Party $\mathcal{A}$:
   $a' \Leftarrow \langle a_{2\ell} - r \rangle$ // Decrypt
   $2^{-\ell}(a' - (a' \bmod 2^{\ell})) \Rightarrow \langle a'' \rangle$ // Cut $\ell$ least significant bits and encrypt
   Send $\langle a'' \rangle$ to $\mathcal{B}$
3: Party $\mathcal{B}$:
   Let $r' = 2^{-\ell}(r - (r \bmod 2^{\ell}))$
   $r' \Rightarrow \langle r' \rangle$
   Obtain $\langle a_{\ell} \rangle$ as $\langle a_{\ell} \rangle = \langle a'' + r' \rangle$
```

A sketch of such a probabilistic rounding protocol can be found in Protocol 3. The protocol is given the two party setting based on SMC using homomorphic encryption. Party $\mathcal{B}$ holds a value $\langle a_{2\ell} \rangle$ with a fractional part of bit length 2ℓ, and should obtain an encryption of the same value where the fractional part is cut and rounded to a precision of ℓ bits.

First, party $\mathcal{B}$ sends an additively blinded version of the value $a_{2\ell}$ to party $\mathcal{A}$, who subsequently decrypts. Next, party $\mathcal{A}$ cuts off the least significant ℓ bits, and sends the result back to $\mathcal{B}$. $\mathcal{B}$ adds the most significant bits of r to the value a'' in order to obtain a_ℓ. This third step has a chance of introducing a 1-bit rounding error, i.e. the protocol has a chance of returing $a_\ell + 1$ instead of a_ℓ; the error occurs if $(a_{2\ell} - r) \bmod 2^\ell \neq (a_{2\ell} \bmod 2^\ell) - (r \bmod 2^\ell)$.

Protocol 4 Deterministic Rescaling Protocol

```
Input: $\mathcal{B}$: $\langle a_{2\ell} \rangle$ with $2\ell$-bit quantization
Output: $\mathcal{B}$: $\langle a_\ell \rangle$ with $\ell$-bit quantization
1: Party $\mathcal{B}$:
   Choose random values $r_1, r_2$.
   $\langle a' \rangle = \langle a_{2\ell} - r_1 \rangle$ // Additively blind the value $\langle a \rangle$ with value $-r_1$
   Create a Garbled Circuit $C$ which computes
       Garbled Circuit C
           Input: $a'$
               $a'' = a' + r_1$
               $\tilde{a} = r_2 + 2^{-\ell}(a'' - (a'' \bmod 2^\ell))$
           Output: $\tilde{a}$
   Send $\langle a' \rangle$ and $C$ to party $\mathcal{A}$.
2: Party $\mathcal{A}$:
   $a' \Leftarrow \langle a' \rangle$ // Decrypt
   Obtain a bit decomposition of $a'$
   Evaluate $C$ on $a'$, obtain result $\tilde{a} = \{C\}_{GC}$.
   $\tilde{a} \Rightarrow \langle \tilde{a} \rangle$ // Encrypt
   Send $\langle \tilde{a} \rangle$ to $\mathcal{B}$.
3: Party $\mathcal{B}$:
   Obtain $\langle a_\ell \rangle$ as $\langle a_\ell \rangle = \langle \tilde{a} - r_2 \rangle$
```

4.2.2 Deterministic Rounding Protocol

The second class of protocols does not introduce probabilistic rounding errors, and thus can be used for applications involving arbitrarily many consecutive multiplications. However, this comes with a drawback in terms of complexity.

Protocol 4 depicts a sketch of the deterministic rescaling protocol presented in [49]. What makes the protocol expensive for use in SMC applications, is the large garbled circuit which has to be sent over the network. For large parameters ℓ, the circuit can easily take the size of a few hundred kilobytes up to some megabytes, making this approach very unattractive for secure computations. In addition to this, the size of the homomorphic encryptions grows very fast (see Chapter 7 for details), making the public key operations computationally too complex.

As it will be seen in Chapter 7, this approach for computing with non-integer values is very efficient for small values ℓ, e.g. when representing values that come with small fractional part. For larger values ℓ, a completely different computational approach is required.

Chapter 5

Secure Computations using Logarithmic Encoding

As we have seen in Chapters 3 and 4 it is possible to efficiently perform secure computations on integer values, which implies the possibility to perform secure and efficient computations on values stored in a fixed point representation with a small fractional part. In the past decade a number of applications using SMC have shown that secure computations based on these techniques can indeed be used in real world applications, with good performance. Recent examples can be found in the area of private auctions [55], biometric computations [36] and privacy-preserving data mining [56].

Most of the above mentioned applications only consider scenarios where computations are performed using integer or Boolean arithmetics. Applications which require to perform computations on non-integer data, such as floating point values, have rarely been presented. In fact, a thorough investigation of techniques that would allow for secure computations on non-integer values was missing.

In this chapter we close this gap and present a computational framework which allows two semi-honest parties to perform secure computations on non-integer values which come from a bounded real interval $\mathcal{D}_\gamma = [-\gamma; \gamma]$. Our solution provides *numerical stability*: Computations can be performed on arbitrarily small or large numbers with finite precision, but yield to results as if they were computed with standard floating-point arithmetic. While in this chapter we refrain from implementing a standardized floating point encoding, we do use a representation that shares the most important properties of floating point arithmetic, i.e. values in $\mathcal{D}_\gamma$ close to zero are represented with higher accuracy than larger values, yielding to a bounded relative error, regardless whether representing very small or very large values.

5.1 Private Function Evaluation

In this section we describe a two-party protocol which allows to evaluate a complex function $f : \mathcal{X} \to \mathcal{Y}$ with $\mathcal{X}, \mathcal{Y} \subset \mathbb{N}$ and $\mathcal{X} = [x_{low}; x_{up}]$ in a private way. This primitive, which will be used as a central building block in our framework, allows to evaluate f on an encryption $\langle x \rangle$ of a value $x \in \mathcal{X}$, producing an encryption $\langle f(x) \rangle$, while keeping x and $f(x)$ secret from both parties. In particular, this will be achieved by performing an oblivious table lookup. The table will be denoted by $\mathcal{T} = (x_i, f(x_i))_{x_i \in \mathcal{X}}$; we use $\mathcal{T}(x)$ to denote the entry of the table $\mathcal{T}$ at position x. The construction will be given in the aforementioned two party scenario, where one party $\mathcal{A}$ holds the private key for some homomorphic encryption scheme, and another party $\mathcal{B}$ holds an encryption $\langle x \rangle$ of a value x and learns an encryption $\langle \mathcal{T}(x) \rangle$ of $\mathcal{T}(x)$. Neither party is allowed to learn the value x nor the plain table entry $\mathcal{T}(x)$.

We will investigate two different approaches to implement this operation: The first approach is based on Oblivious Pseudo-Random Function Evaluation (OPRF), while the second approach is based on Oblivious Transfer (OT). As we will see, the first approach achieves a lower communication complexity than the second approach for small table sizes. While the second approach requires slightly more communication, the clear advantage is its low computational complexity.

Both approaches allow for adaptive protocols, that re-use the table $\mathcal{T}$ for various look-up operations. This results in less communication and computational complexity, as complexity can be amortized over a certain amount of operations, yielding to a much better practical performance. In particular this means that the table has to be generated and transferred only once, and then can be used for a variable number of times. Adaptive protocols still hide the values x and $f(x)$, at the price that party $\mathcal{B}$ can learn whether a value was queried more than once. However, party $\mathcal{B}$ will have no information which value was queried more than once; furthermore, $\mathcal{B}$ will not learn any information on values queried only once. Depending on the application scenario, this may be acceptable if this event happens rarely.

5.1.1 OPRF Construction

This construction is based on OPRFs introduced in [22]. In this section we assume that all values are encrypted in the Paillier encryption scheme, thus messages come from the plaintext space $\mathbb{Z}_n$ for some RSA modulus n (see Section 3.4). The protocol consists of two phases: one initialization and one evaluation phase. In the initialization phase, $\mathcal{A}$ generates a prime number $\mathfrak{p}$ in a way that $n \mid (\mathfrak{p}-1)$ and an element $\mathfrak{g}$ of order n in $\mathbb{Z}_\mathfrak{p}$. $\mathcal{A}$ further selects a random element $k \in \mathbb{Z}_n$. These choices assure that $\mathfrak{g}_x = \mathfrak{g}^{\frac{1}{k+x}} \pmod{\mathfrak{p}}$ is a pseudo-random function [24].

Let again $\mathcal{X} = [x_{low}; x_{up}]$ be the set of numbers on which f operates on. $\mathcal{A}$ prepares a table $\mathcal{T}$ with two columns, where the first column contains the value $\mathfrak{g}_x = \mathfrak{g}^{\frac{1}{k+x}}$ and the second column contains an encryption of $f(x)$ for each $x \in \mathcal{X}$. Furthermore $\mathcal{A}$ permutes the table by sorting in ascending order according to the first row. Now $\mathcal{A}$ sends the table $\mathcal{T}$ together with an encryption of k used in the function $\mathfrak{g}^{\frac{1}{k+x}}$ to party $\mathcal{B}$. The protocol for the initialization is depicted in Protocol 5.

Protocol 5 Computing $f(x)$ using OPRFs – Initialization Step

Input: Party $\mathcal{B}$: $[x], [k], \mathcal{T}$

Output: Party $\mathcal{B}$: $[f(x)]$

```
1: Party $\mathcal{A}$:
   Choose $k \in_R \mathbb{Z}_n$
   for each $x \in \mathcal{X}$
      Add $\mathfrak{g}_x = \mathfrak{g}^{\frac{1}{k+x}}$ and $[f(x)]$ to table $\mathcal{T}$
   end
   Send $[k]$ and sorted table $\mathcal{T}$ to party $\mathcal{B}$
```

In the evaluation phase, $\mathcal{B}$ obliviously learns an encryption $[f(x)]$ for some value $[x]$ (obliviously to himself and to party $\mathcal{A}$), given $[k]$ and $\mathcal{T}$, by running Protocol 6. $\mathcal{A}$ and $\mathcal{B}$ obliviously evaluate the pseudo-random function $\mathfrak{g}_x = \mathfrak{g}^{\frac{1}{k+x}}$ on $[x]$, so that the result will be available in the clear only to $\mathcal{B}$ at the end of the protocol. This knowledge allows $\mathcal{B}$ to "look up" the desired result in the table received before by selecting the row of $\mathcal{T}$ that contains an encryption of $f(x)$. As party $\mathcal{B}$ does not know the key k in the clear, he cannot access other encryptions in the table, since he cannot look up their positions in $\mathcal{T}$.

Since the value $\mathfrak{g}_x$ is pseudo random, the proposed approach allows to use the same table $\mathcal{T}$ prepared in the initialization phase for multiple queries. Thus, the complexity of generating $\mathcal{T}$ can be amortized over a certain number of operations, yielding to an adaptive protocol with good practical performance. However, if $\mathcal{T}$ is used multiple times, there is a chance that a single value will be requested more than once. This information will be leaked to party $\mathcal{B}$. To limit this information flow, the table $\mathcal{T}$ can be updated periodically after a number of queries have been performed. To this end, party $\mathcal{A}$ re-runs the initialization phase and provides party $\mathcal{B}$ with the new table and a new key k'.

Protocol 6 Computing $f(x)$ using OPRFs – Evaluation Step

Input: Party $\mathcal{B}$: $[x]$ with $x \in \mathcal{X}$

Output: Party $\mathcal{B}$: $[f(x)]$

```
1: Party $\mathcal{B}$:
   Let $r \in_R \mathbb{Z}_n$ in $[y] := [r(x+k)] = ([x][k])^r$
   Send $[y]$ to party $\mathcal{A}$
2: Party $\mathcal{A}$:
   $y = \text{Decrypt}([y])$
   $\mathfrak{g}_y = \mathfrak{g}^{(y^{-1})}$
   Send $\mathfrak{g}_y$ to party $\mathcal{B}$
3: Party $\mathcal{B}$:
   $\mathfrak{g}_x = \mathfrak{g}_y^r$
   Look up entry $\mathfrak{g}_x$ in $\mathcal{T}$ and obtain $[f(x)]$
```

5.1.1.1 Optimizations

As $\mathcal{T}$ can become very large, depending on the choices made for f and $\mathcal{X}$, we describe several optimizations which allow to decrease the size of $\mathcal{T}$.

Table index. As each $\mathfrak{g}_x$ is a random value of approximately 1024 bits, but $\mathcal{T}$ has at most $|\mathcal{T}| \ll 2^{1024}$ entries, it is sufficient to choose only the least significant bits of $\mathfrak{g}_x$ as an index. For example, for realistic examples such as $|\mathcal{T}| = 2^{10}$ it suffices to choose the 32 least significant bits. This results in a great reduction of communication. In the rare case that a collision occurs, we suggest to either use more bits for the entries with a collision, or choose a new parameter k and restart the initialization.

Statistical hiding. The size of the second column can be reduced as well. Let $\omega, \kappa > 0$ and $r_0, \dots, r_\kappa$ be random ω-bit numbers, where ω is a security parameter. Rather than storing an encryption of the value $f(x)$ in the table $\mathcal{T}$, it suffices to store a value $f(x) + s_x$, where $s_x = \sum_{i=0}^{\kappa} \alpha_i \cdot r_i$ is a linear combination of random values $r_0, \dots, r_\kappa$. If we provide party $\mathcal{B}$ with encryptions $[r_i]$, and if the values α_i are chosen pseudo-randomly, i.e., by extracting them as random chunks (of small bit-length) of the value $\mathfrak{g}_x$, then party $\mathcal{B}$ can reconstruct the value $f(x)$ by encrypting the table entry in row with index $\mathfrak{g}_x$ and then subtracting $[s_x] = [\sum_{i=0}^{\kappa} \alpha_i r_i]$. As long as party $\mathcal{B}$ does not access more than κ blinded values from table $\mathcal{T}$ (which could give him a system of linear equations of the form $\sum_{i=0}^{\kappa} \alpha_i \cdot r_i$), the values r_i remain hidden and thus also the values $f(x)$ remain statistically hidden from $\mathcal{B}$.

5.1.1.2 Security

In this section we sketch a security proof for the protocol for private function evaluation presented in Protocol 6. Security for larger protocols, such as the protocols from Sections 5.2.4, 8.3 and 8.4, follows from the composition theorem for the semi-honest model [10, Theorem 7.3.3].

We consider the security of Protocol 6 for computing an encryption of $[f(x)]$. Both input and output of party $\mathcal{A}$ are $\perp$. The inputs of party $\mathcal{B}$ are $([x], [k], \mathcal{T})$, while as output he learns $[f(x)]$. Recall that we have a homomorphic encryption scheme, where party $\mathcal{A}$ knows the public and private key and party $\mathcal{B}$ knows the public key. In addition to this, both parties know a public value $\mathfrak{g}$, which is a generator of a subgroup of some finite field $\mathbb{Z}_{\mathfrak{p}}$. The evaluation of the sub-protocol is as follows:

- Party $\mathcal{B}$ generates a random r and sends $[y] = [r(x + k)]$ to party $\mathcal{A}$.
- Party $\mathcal{A}$ decrypts $[r(x + k)]$, computes $r(x + k)$ and sends $v = g^{\frac{1}{r(x+k)}}$ to party $\mathcal{B}$.[1]
- Party $\mathcal{B}$ computes $v^r = g^{\frac{1}{k+x}}$ and obtains an encryption $[f(x)]$ by selecting entry $g^{\frac{1}{k+x}}$ from table $\mathcal{T}$.

Next we prove that the protocol given above privately computes $(\perp, [x]) \to (\perp, [f(x)])$. Party $\mathcal{A}$'s view $\text{VIEW}_{\mathcal{A}}(\perp)$ is $\{r(x + k)\}$. Here, $\text{VIEW}_{\mathcal{A}}(\perp)$ can be simulated by a PPTA because $r(x + k)$ is uniformly distributed in $\mathbb{Z}_n$ and is therefore statistically indistinguishable from a value chosen uniformly at random from $\mathbb{Z}_n$. Therefore, the views are statistically indistinguishable which implies that they are also computationally indistinguishable (see [10], Section 7.2.1.2).

Party $\mathcal{B}$'s view $\text{VIEW}_{\mathcal{B}}([x], [k], \mathcal{T})$ consists of $([x], [k], \mathcal{T}, r, g^{\frac{1}{r(k+x)}}, g^{\frac{1}{k+x}}, [f(x)])$. To simulate, first construct a new view H_1 by replacing $[x]$ and $[k]$ by $[x']$ and

[1]Technically if $gcd(r(x + k), n) \neq 1$, the protocol aborts (where n is the underlying modulus of the group).

$[k']$ in $\textsc{View}_{\mathcal{B}}([x])$, where x', k' are randomly chosen. Semantic security of the Paillier encryption scheme guarantees that the two views $\textsc{View}_{\mathcal{B}}([x])$ and H_1 are indistinguishable. Subsequently, construct a new view H_2 by replacing r, $\mathfrak{g}^{\frac{1}{r(k+x)}}$ and $\mathfrak{g}^{\frac{1}{k+x}}$. First, we replace r by r' in H_1, where r' is randomly chosen. The value $g^{\frac{1}{r(k+x)}}$ can be replaced by $\mathfrak{g}^{\frac{1}{r'(k'+x')}}$, where k' and x' are chosen uniformly at random from $\mathbb{Z}_n$. Furthermore, we replace $\mathfrak{g}^{\frac{1}{k+x}}$ with $\mathfrak{g}^{\frac{1}{k'+x'}}$. Views H_1 and H_2 are indistinguishable because $(\mathfrak{g}^{\frac{1}{r(k+x)}}, \mathfrak{g}^{\frac{1}{k+x}}, r)$ and $(\mathfrak{g}^{\frac{1}{r'(k'+x')}}, \mathfrak{g}^{\frac{1}{k'+x'}}, r')$ are identically distributed. This is due to the fact that $\mathfrak{g}^{\frac{1}{r(k+x)}}$ implements a pseudorandom function [24]. Finally, we create a view H_3 where we replace $\mathcal{T}$ and $[f(x)]$ with a new table $\mathcal{T}'$ and a value $[f(x')]$. The table $\mathcal{T}'$ is constructed as follows: Create a table $\mathcal{T}''$ which contains values $\mathfrak{g}^{\frac{1}{k''+x''}}$ in the first colum, and encryptions $[f(x'')]$ in the second column, where the values k'' and x'' are random elements. Let m be an integer chosen uniformly at random from $\{1, \ldots, |\mathcal{T}|\}$. Replace the m-th row of $\mathcal{T}''$ with $(\mathfrak{g}^{\frac{1}{k'+x'}}, [f(x')])$, where k' and x' are the random values used to construct view H_1, H_2.

Now, the two views H_3 and $\textsc{View}_{\mathcal{B}}([x], [k], \mathcal{T})$ are computationally indistinguishable. Note further that H_3 can be generated from $([x], g^{\frac{1}{k+x}})$ by a PPTA. This shows that both views can efficiently be simulated.

5.1.2 OT Construction

In this section we will show how the table look-up operation can significantly be improved by reducing the computational complexity, while accepting a slightly increased communication complexity. In this section, when referring to a homomorphic encryption, we exclusively use the DGK cryptosystem (see Section 3.4).

Creating and transmitting the full table in Section 5.1.1 is rather costly. Therefore, we transform the lookup in a large table of size $|\mathcal{T}|$ into two lookups in smaller tables of size $t = \sqrt{|\mathcal{T}|}$ and one oblivious evaluation of some polynomial. In the remainder of this section we assume that x is a positive value; this can always be achieved by shifting the interval $\mathcal{X} = [x_{low}; x_{up}]$ to $\mathcal{X}' = [0; x_{up} - x_{low}]$. For simplicity of presentation, we will further assume that the table $\mathcal{T}$ has 2^{2d} entries, thus $t = \sqrt{|\mathcal{T}|} = 2^d$. We will later relax this assumption. For a good performance, we choose the parameter u of the DGK cryptosystem as the smallest prime number such that $u > |\mathcal{T}|$.

We first split x into two values y and z which consist of the d most significant resp. least significant bits of x, i.e. $x = y \cdot t + z$. Now, we perform two simultaneous

y	$\tilde{x}$
0	$\tilde{x}_0$
1	$\tilde{x}_1$
$\vdots$	$\vdots$
$t-1$	$\tilde{x}_{t-1}$

Table 5.1: Table lookup for value y.

table lookups. In the first lookup (e.g. see Table 5.1), the value y is mapped to a value $\tilde{x} \in \{\tilde{x}_0, \ldots, \tilde{x}_{t-1}\}$, where the values $\tilde{x}_0, \ldots, \tilde{x}_{t-1}$ are chosen at random from $\mathbb{Z}_u$. Next, polynomials P_z are generated in a way that on the evaluation points $\tilde{x}_0, \ldots, \tilde{x}_{t-1}$ the polynomials take on values $f(x_i)$, i.e. $P_z(\tilde{x}_y) = f(y \cdot t + z)$.

z	P_z
0	P_0
1	P_1
$\vdots$	$\vdots$
$t-1$	P_{t-1}

Table 5.2: Table lookup for value z.

In the second lookup (see Table 5.2), the value z is used to select the polynomial P_z which is finally evaluated on $\tilde{x}$ to obtain the result $[\![f(x)]\!] = [\![f(y \cdot t + z)]\!] = [\![P_z(\tilde{x})]\!]$ (see Table 5.3).

	$\tilde{x}_0$	$\tilde{x}_1$	$\ldots$	$\tilde{x}_{t-1}$
$P_0(\tilde{x})$	$f(x_1)$	$f(x_2)$	$\ldots$	$f(x_t)$
$P_1(\tilde{x})$	$f(x_{t+1})$	$f(x_{t+2})$	$\ldots$	$f(x_{2t})$
$\vdots$	$\vdots$			
$P_{t-1}(\tilde{x})$	$f(x_{\lvert\mathcal{T}\rvert-t+1})$	$f(x_{\lvert\mathcal{T}\rvert-t+2})$	$\ldots$	$f(x_{\lvert\mathcal{T}\rvert})$

Table 5.3: New table lookup.

Thus, the full protocol (see Protocols 7 and 8) consists of the following steps, which will be described in detail in the subsequent sections:

- (Offline): Prepare representation as polynomials (Step 1).

- Extract the bits of $[\![x]\!]$ to obtain values y, z, with $x = y \cdot t + z$ (Steps 2 and 3).
- Transfer values $\tilde{x}$ and P_z with tables of size t using OT (Steps 4 and 5).
- Reconstruct the value $[\![\mathcal{T}(x)]\!]$ (Step 6).

5.1.2.1 Representation as polynomials

We first show how values $\tilde{x}_i$ and polynomials P_j, with $i, j \in \{0, \dots, t-1\}$, should be chosen in order to allow an efficient reconstruction of an encryption of the desired value $f(x)$. This step will be done by party $\mathcal{A}$ in some initialization phase (which could be performed offline).

In our protocols, party $\mathcal{B}$ holds an encryption $[\![x]\!]$ whereas party $\mathcal{A}$ assists in the computations. Party $\mathcal{A}$ will therefore choose polynomials and values $\tilde{x}_i$. Initially, party $\mathcal{A}$ chooses values $\tilde{x}_0, \dots, \tilde{x}_{t-1}$ uniformly at random from $\mathbb{Z}_u$ (with $\tilde{x}_i \neq \tilde{x}_j$ for $i \neq j$). Each of these values will be assigned to one value $y \in \{0, \dots, t-1\}$. Choosing these values at random allows to later reveal some values $\tilde{x}_i$ to party $\mathcal{B}$ as long as the mapping $\{0, \dots, t-1\} \to \{\tilde{x}_0, \dots, \tilde{x}_{t-1}\}$ remains hidden. Revealing these values to party $\mathcal{B}$ is crucial for the performance of our solution, as this allows party $\mathcal{B}$ to compute powers $\tilde{x}^j$ without interaction with party $\mathcal{B}$. Next, the polynomials P_z will be determined using polynomial interpolation. For each value $z \in \{0, \dots, t-1\}$, party $\mathcal{A}$ computes coefficients $a_0, \dots, a_t$ of a polynomial $P_z(x) = \sum_{j=0}^{t} a_j x^j$ in a way that $P_z(\tilde{x}_j) = f(tj + z)$. Since the polynomials will be chosen of degree t, there will be one degree of freedom left. This will be used to add additional randomness to the process, thus we demand that $P_z(\tilde{x}_t) = r_t$ for values $\tilde{x}_t \notin \{\tilde{x}_0, \dots, \tilde{x}_{t-1}\}$ and r_t which are randomly chosen for each polynomial P_z. Note that, due to the random choice of the interpolation points $\tilde{x}_i$, also the coefficients a_j appear to be randomly chosen from $\mathbb{Z}_u$.

Before submitting a polynomial P_z to party $\mathcal{B}$, we encrypt some of the coefficients. Encrypting some of the coefficients a_j will allow party $\mathcal{B}$ merely to compute an *encryption* of the desired value $[\![\mathcal{T}(x)]\!]$. We therefore choose $\kappa+1$ random values $\alpha_0, \dots, \alpha_\kappa \in_R \mathbb{Z}_u$ and replace the first $\kappa + 1$ coefficients $a_0, \dots, a_\kappa$ of each polynomial P_z by $\beta_j := a_j + \alpha_j \bmod u$. A full representation of a polynomial P_z then consists of the values $\beta_0, \dots, \beta_\kappa, a_{\kappa+1}, \dots, a_t$ and encryptions $[\![\alpha_0]\!], \dots, [\![\alpha_\kappa]\!]$. Note that to prevent party $\mathcal{B}$ from guessing the coefficients $a_0, \dots, a_\kappa$, we need to make sure that $\kappa + 1 \geq \omega / \log_2 u$ (where ω denotes a statistical security parameter). For example, for $\omega = 80$ bit and u being 16 bit, we require $\kappa \geq 4$.

For practical implementations, this is typically already the case, since a set of polynomials P_z and evaluation points $\tilde{x}_i$ can (and should, for efficiency reasons) be reused κ times. However, if party $\mathcal{B}$ learns more than κ values $\tilde{x}_i$ and if κ is small, he can guess the result of the polynomial evaluation, e.g. test all values $P_z(x) = \sum_{j=0}^{t} a_j x^j = v$, with $v \in \mathbb{Z}_u$, and then solve the system of linear equations for the values $\alpha_0, \ldots, \alpha_\kappa$. Thus, in order to prevent this kind of attack, we allow each set of polynomials to be used at most κ times; once this happens, a new set of polynomials must be prepared. Note that, naturally party $\mathcal{A}$ could as well encrypt all coefficients. However, more encrypted coefficients means more communication and computation during the initialization, thus we only encrypt $\kappa + 1$ coefficients.

Protocol 7 Optimized table look-up (Part 1)

```
Input: Party $\mathcal{B}$: $[\![x]\!]$
Output: Party $\mathcal{B}$: $[\![\mathcal{T}(x)]\!]$
   Initialization:
1: Party $\mathcal{A}$: Generate values $\tilde{x}_i$ and polynomials $P_i$, prepare tables $\mathcal{U}_1, \mathcal{U}_2, \mathcal{P}$
   (indices, values $\tilde{x}_i$ and list of encrypted polynomials) and send $[\![\alpha_0]\!], \ldots, [\![\alpha_\kappa]\!]$
   and $\mathcal{P}$ to party $\mathcal{B}$
   Evaluation:
2: Party $\mathcal{B}$: Choose $r \in_R \mathbb{Z}_u$
   $r_1, r_2 \in_R \{0, \ldots, t-1\}$.
   $[\![a]\!] = [\![x + r]\!]$
   Construct a garbled circuit $C$ which realizes the following functionality:
      Garbled Circuit C
          Input: $a$
            $b_1 := a - r$, $b_2 := a - r + u$
            if ($b_1 < 0$) then
              $b := b_2$
            else
              $b := b_1$
            end
            Set $z := b \bmod t$, $y := (b - z)/t$
            $o_1 = (y - r_1) \bmod t$
            $o_2 = (z - r_2) \bmod t$
          Output: $o_1, o_2$
   Send $C$ and $[\![a]\!]$ to party $\mathcal{A}$
```

5.1.2.2 Detailed protocol description

The oblivious table look-up is presented in Protocols 7 and 8; it consists of the following steps:

Initialization. In a first step, party $\mathcal{A}$ computes values $\tilde{x}_i$, polynomials and values α_i. Next, party $\mathcal{A}$ prepares tables $\mathcal{U}_1$,$\mathcal{U}_2$ and a list $\mathcal{P}$ which will be used in the oblivious table look-ups. The list $\mathcal{P}$ contains all polynomials in random order. Table $\mathcal{U}_1$ contains the values $\tilde{x}_i$ which is selected by value y (see Table 5.1). Table $\mathcal{U}_2$ is used to transfer the index of a polynomial in $\mathcal{P}$ (the value z in Table 5.2). Both the list $\mathcal{P}$ and values $[\![\alpha_0]\!], \ldots, [\![\alpha_\kappa]\!]$ can be transferred at any time (e.g. ahead of the evaluation phase). Tables $\mathcal{U}_1$ and $\mathcal{U}_2$ will then be used in steps 3 and 4 of the protocol.

Split the value x, compute y, z. In steps 2 and 3 of Protocols 7 and 8, party $\mathcal{B}$ holds an encryption $[\![x]\!]$ and should obtain two values of y, z such that $x = ty + z$. For efficiency reasons, at this point we use garbled circuits. The straightforward alternative at this point would be to keep the values under homomorphic encryption, and perform all binary operations on the encrypted bits. However, garbled circuits yield to much better performance for these kind of operations. We further use the extended oblivious transfer protocol [57] in order to transfer large chunks of symmetric keys for the garbled circuits. The extended oblivious transfer protocol builds on the OT protocol presented in [26]. This is, party $\mathcal{A}$ throws random coins and randomly requests garbled circuit keys according to these coins. These keys will later then be used in Protocol 7 to construct the circuit. Transferring the keys beforehand can be done this way, since the distribution of the inputs to our circuits is close to values with random bits (in fact, they are random values from $\mathbb{Z}_u$). In the circuit, we first remove the additional blinding factor, by subtracting r from the input a. This computation is performed modulo u, and is realized as two addition circuits in binary 2th complement representation. A multiplexer is used to select the correct value b_1 or b_2 according to the sign bit of b_1. The d least significant bits of b are then output as value o_2 under another additive blinding, while the d most significant bits of b are output as value o_1.

Implementing the table lookups. After Step 3 of Protocol 8, party $\mathcal{A}$ holds the values y and z, additively blinded with values r_1, r_2 which are known to party

Protocol 8 Optimized table look-up (Part 2)

```
   Protocol 7 continued:
3: Party $\mathcal{A}$:
   $a = \text{Decrypt}([\![a]\!])$
   Evaluate the circuit $C$ on input $a$, obtain outputs $o_1, o_2$
   for $i = 1, 2$
      // if $i = 1$ send value $\tilde{x}$ (table $\mathcal{U}_1$)
      // if $i = 2$ transfer index $\pi(i)$ (table $\mathcal{U}_2$)
4: Party $\mathcal{A}$:
      // Prepare the rotated list for table $\mathcal{U}_i$:
      $m_0 = \mathcal{U}_i(o_i + 0 \bmod t)$
      $m_1 = \mathcal{U}_i(o_i + 1 \bmod t)$
      ...
      $m_{t-1} = \mathcal{U}_i(o_i + (t-1) \bmod t)$
5: Party $\mathcal{A}$:
      Request value $m_{r_i}$ using OT
   end
   // Party $\mathcal{B}$ now holds polynomial $P_z$ and a value $\tilde{x}$
6: Party $\mathcal{A}$:
   for $i \in \{0, \dots, \kappa\}$
      $[\![a_i \tilde{x}^i]\!] = [\![\alpha_i + \beta_i]\!]^{\tilde{x}^i}$
   end
   for $i \in \{\kappa + 1, \dots, t\}$
      $[\![a_i \tilde{x}^i]\!] = \text{Encrypt}(a_i \tilde{x}^i)$
   end
   $[\![\mathcal{T}(x)]\!] = [\![P_z(\tilde{x})]\!] = [\![\sum_{i=0}^{t} a_i \tilde{x}^i]\!]$
```

$\mathcal{B}$ (thus $y = (o_1 + r_1) \bmod t$ and $z = (o_2 + r_2) \bmod t$). The goal is now to let party $\mathcal{B}$ learn the values P_z and $\tilde{x}$, without revealing y or z.

To this end, Steps 4 and 5 in Protocol 8 depict how to extend a blackbox version of OT to allow for additively blinded values as inputs (e.g. see [58, 59]). Party $\mathcal{A}$ first rotates the tables $\mathcal{U}_i$ according to the values o_i ($i \in \{1, 2\}$). The rotated lists will then be used as input for some OT protocol to obtain the correct output. When party $\mathcal{B}$ requests the values r_i, he clearly obtains the correct result, since e.g. for $i = 1$ the value at position r_1 is table entry $\mathcal{U}_1(o_1 + r_1) = \mathcal{U}_1(y - r_1 + r_1) = \mathcal{U}_1(y) = \tilde{x}_y$.

Since we allow each set of polynomials to be used κ times, we do not use the polynomials as direct input to the OT protocol. Instead, party $\mathcal{A}$ uses a permutation π to permute the set of polynomials $P_0, \ldots, P_{t-1}$ and transfers this list $\mathcal{P} := P_{\pi(0)}, \ldots, P_{\pi(t-1)}$ to party $\mathcal{B}$ during the initialization phase. Party $\mathcal{B}$ will stay oblivious about the correct order of the polynomials, since the coefficients of the polynomials appear to be random elements of $\mathbb{Z}_u$. As input for the OT protocol, party $\mathcal{A}$ uses the permuted list of indices $\pi(0), \ldots, \pi(t-1)$. This allows party $\mathcal{B}$ to learn the correct index $\pi(z)$ of polynomial P_z in the list $\mathcal{P}$. Using the same set of polynomials multiple times significantly reduces the communication overhead. At the same time party $\mathcal{B}$ has a small chance to see if there was one value which was queried more than once (however, party $\mathcal{B}$ will get no information which value was actually queried more than once).

As it was explained before, the parties will run Steps 4 and 5 of Protocol 8 two times. One time with input o_1 and table $\mathcal{U}_1 = (i, \tilde{x}_i)_{i \in I}$ to obtain a value $\tilde{x}$, and a second time with input o_2 and table $\mathcal{U}_2 = (i, \pi(i))_{i \in I}$ (with $I = \{0, \ldots, t-1\}$) which allows $\mathcal{B}$ to select the right polynomial P_z from the permuted list $\mathcal{P}$. A polynomial P_z will be represented by the values $\beta_0, \ldots, \beta_\kappa, a_{\kappa+1}, \ldots, a_t$; thus, we refer to a polynomial P_z as the concatenation of the values $\beta_0, \ldots, \beta_\kappa, a_{\kappa+1}, \ldots, a_t$.

Reconstructing the value $[\![\mathcal{T}(x)]\!]$. After party $\mathcal{B}$ has learned a value $\tilde{x}$ and an encrypted polynomial $\tilde{P}_y$ he needs to reconstruct the value $[\![\mathcal{T}(x)]\!]$. Party $\mathcal{B}$ first computes encryptions of the coefficients of P_z, i.e. $[\![a_i]\!] = [\![\alpha_i + \beta_i]\!]$ for $i \in \{0, \ldots, \kappa\}$, by using the homomorphic property. Next, he encrypts the remaining coefficients $a_{\kappa+1}, \ldots, a_k$. Finally, he computes $[\![\mathcal{T}(x)]\!] = [\![P_z(\tilde{x})]\!] = [\![\sum_{i=0}^{t} a_i \tilde{x}^i]\!] = \prod_{i=0}^{t} [\![a_i]\!]^{x^i}$. Note that the encryptions in Step 6 of Protocol 8 can be performed very efficiently. Only one of these encryptions needs to be (re-)randomized, while for the remaining ones it suffices to compute a deterministic encryption (see Section 2.2.3.3).

Dynamic table size. Naturally, the size of table $\mathcal{T}$ does not need to be restricted to 2^{2d}. In fact our scheme allows arbitrary table sizes. In this case, we compute $t_1 = \lfloor 0.5 \log_2 |\mathcal{T}| \rfloor$, and choose a minimal t_2 such that $t_1 t_2 \geq |\mathcal{T}|$. Party $\mathcal{A}$ now computes t_1 different polynomials of degree t_2. It is straightforward to modify the rest of Protocols 7 and 8 accordingly.

5.2 Secure Computations on Non-Integer Values

In this section we present a framework which allows two parties to perform secure computations on values from some interval $\mathcal{D}_\gamma = [-\ell; +\ell] \subset \mathbb{R}$. Note that we describe the framework in its full generality; in case that computations are performed only on a limited set of numbers (e.g., only on probabilities in the interval $[0; 1]$) further optimizations are possible.

5.2.1 Data Representation

We start by giving a description of how values from the real domain $\mathcal{D}_\gamma$ will be represented. We require the representation to meet the following conditions:

- First, the scheme should allow a representation as integer values, in order to use efficient methods from generic Secure Multiparty Computation for some sub-problems.
- Second, the representation should have a similar behavior as floating point arithmetic: the precision for numbers close to 0 should be higher than for numbers far away from 0. This is desirable in order to achieve a bounded relative error when representing both very small and rather big numbers.
- The data encoding should allow us to perform certain arithmetic operations directly on the encoded values.

Our solution to this problem builds on the work of Kingsbury and Rayner [60]. We use a logarithmic representation for the numbers in the interval $\mathcal{D}_\gamma = [-\gamma; \gamma]$ with $\gamma > 0$. Let $S \in \mathbb{N}$ be a scaling factor, $B > 0$ a suitable base for some logarithm and C be a constant with $\log_B(C) > 0.5/S + \log_B(\gamma)$. We represent a value $x \in \mathcal{D}_\gamma$ as a tuple $(\rho_x, \sigma_x, \tau_x)$. Here, $\rho_x \in \{0, 1\}$ is a boolean value indicating whether the represented number is zero, i.e., $\rho_x = 1$ if x is not equal to zero and $\rho_x = 0$ otherwise. Similarly, σ_x encodes the sign and will be kept as a boolean, i.e., $\sigma_x = 0$ if $x \geq 0$ and $\sigma_x = 1$ otherwise. If $\rho_x = 1$ we compute an encoding τ_x of the absolute value as $\tau_x = \lceil -S \cdot \log_B(\frac{|x|}{C}) \rfloor$, where $\lceil \cdot \rfloor$ denotes rounding to the nearest integer. Note that the choice of C ensures that $\tau_x > 0$ for all $x \in [-\gamma; \gamma]$. In case of $\rho_x = 0$ both σ_x and τ_x will be set to 1 and can later contain arbitrary values. If we limit the space for the values τ_x to some interval $[1; 2^\delta - 1]$ by storing τ_x as δ-bit number, we can exactly represent a set of $2^{\delta+1} - 1$ distinct numbers from $\mathcal{D}_\gamma$.

We denote the representation of any given $x \in \mathcal{D}_\gamma$ in $\mathcal{L}_\delta := \{0,1\} \times \{0,1\} \times \{1,\dots,2^\delta - 1\}$ by $\overline{x}$. The backward transformation $\mathcal{L}_\delta \to \mathcal{D}_\gamma$ is given by

$$(\rho_x, \sigma_x, \tau_x) \mapsto \rho_x \cdot (1 - 2\sigma_x) \cdot C \cdot B^{(-\tau_x/S)}.$$

5.2.2 Parameters

According to the desired level of accuracy and the size γ for the interval of represented numbers, the parameters S, B, C and δ can be adjusted. We first note that for each pair (S_1, B_1) and a base B_2 there exists a unique S_2 such that $S_1 \log_{B_1}(x) = S_2 \log_{B_2}(x)$ for all $x \in \mathbb{R}^+$. Thus it suffices to consider only one fixed base B and adjust the parameters S, C and δ accordingly.

The relative distance $\Delta x/x$ between two subsequent numbers in $\mathcal{D}_\gamma$, which can exactly be represented in our encoding, is

$$\Delta x/x = \frac{C \cdot B^{-\tau_x/S} - C \cdot B^{-(\tau_x+1)/S}}{C \cdot B^{-\tau_x/S}} = 1 - B^{-1/S}, \tag{5.1}$$

and thus depends only on B and S. To achieve a fixed maximum relative error $\Delta x/x$ it thus suffices to fix a base B and compute S as $S = -(\log_B(1 - \Delta x/x))^{-1}$. For example, for a base $B = 2$ and scaling factor $S = 100$, the maximal relative representation error is given by $6.9 \cdot 10^{-3}$.

Other factors to consider when choosing the parameters are the smallest and the greatest positive value which can be represented; in our encoding we have $C \cdot B^{-(2^\delta - 1)/S}$ for the minimal and $C \cdot B^{-1/S}$ for the maximal positive value. As the scheme represents numbers from the interval $\mathcal{D}_\gamma$, we choose C as $C > \gamma \cdot B^{1/S}$. Thus, for most choices of S and B we can let $C \approx \gamma$. For fixed S, B and C, we can set the parameter δ to adjust the smallest number which can be represented by the scheme. For example, for parameter values of $C = 1, B = 2, S = 100$ and $\delta = 16$ the smallest positive value that can be represented is $5.25 \cdot 10^{-198}$. In addition, δ influences the precision of the arithmetic, i.e., the number of values from $\mathcal{D}_\gamma$ which can be represented exactly.

There are no general guidelines on how to choose the parameters; they must be chosen according to the problem domain. Usually, this process involves an experimental analysis of the propagation of errors during the computation (for an example see Section 8.5).

5.2.3 Arithmetic Operations

We will now describe how to perform basic arithmetic operations on encoded values $\overline{x}, \overline{y} \in \mathcal{L}_\delta$.

5.2.3.1 Multiplication and Division

Due to the logarithmic representation it is quite easy to compute an encoding $\overline{xy}$ of the product $x \cdot y$, given encodings $\overline{x}$ and $\overline{y}$. Let $\tau_{C^2} = \lceil S \cdot \log_B(C) \rfloor$. The function

$$\begin{aligned} \mathbf{LPROD}(\overline{x}, \overline{y}) &= \mathbf{LPROD}((\rho_x, \sigma_x, \tau_x), (\rho_y, \sigma_y, \tau_y)) \\ &= (\rho_x \rho_y, \sigma_x \oplus \sigma_y, \tau_x + \tau_y - \tau_{C^2}) \\ &= \overline{xy} \end{aligned} \tag{5.2}$$

achieves the desired functionality. Note that $\mathbf{LPROD}(\overline{x}, \overline{y}) = \overline{xy}$: If either $x = 0$ or $y = 0$, then also $\overline{xy}$ is an encoding of zero, and if both $x, y \neq 0$ then

$$\begin{aligned} \tau_{xy} &= \tau_x + \tau_y - \tau_{C^2} \\ &= -S \log_B(x/C) - S \log_B(y/C) - S \log_B(C) \\ &= -S \log_B(xy/C). \end{aligned}$$

Dividing two numbers $\overline{x}, \overline{y} \in \mathcal{L}_\delta$ with $y \neq 0$ is similar:

$$\begin{aligned} \mathbf{LDIV}(\overline{x}, \overline{y}) &= \mathbf{LDIV}((\rho_x, \sigma_x, \tau_x), (\rho_y, \sigma_y, \tau_y)) \\ &= (\rho_x, \sigma_x \oplus \sigma_y, \tau_x - \tau_y + \tau_{C^2}) \\ &= \overline{x/y}. \end{aligned} \tag{5.3}$$

5.2.3.2 Addition and Subtraction

Given $\overline{x}$ and $\overline{y}$, computing $\overline{x+y}$ and $\overline{x-y}$, which will be denoted by **LSUM** and **LSUB** in the sequel, is more involved. We first note that in order to compute a subtraction, it suffices to swap the sign σ_y of $\overline{y}$ and then perform an addition. Thus we limit ourselves to describe the addition. Let $\lambda_1, \ldots, \lambda_4 \in \{0, 1\}$, where

$$\lambda_1 = \begin{cases} 1, & \text{if } \tau_x < \tau_y \\ 0, & \text{otherwise,} \end{cases} \tag{5.4}$$

$$\lambda_2 = \begin{cases} 1, & \text{if } \tau_x = \tau_y \\ 0, & \text{otherwise,} \end{cases} \tag{5.5}$$

$$\lambda_3 := \sigma_x \oplus \sigma_y, \text{ and} \tag{5.6}$$

$$\lambda_4 := \rho_x \rho_y. \tag{5.7}$$

The new value ρ_{x+y} can be obtained by

$$\rho_{x+y} = \neg((\lambda_2\lambda_3) + \neg(\rho_x + \rho_y)). \tag{5.8}$$

Furthermore,

$$\begin{aligned} \sigma_{x+y} &= \mathrm{MUX}_{\lambda_2}(\rho_y\sigma_y + \rho_x\sigma_x, t), \text{ with} & (5.9)\\ t &= \mathrm{MUX}_{\lambda_1}(\rho_x\sigma_x + (\neg\rho_x)\sigma_y, \rho_y\sigma_y + (\neg\rho_y)\sigma_x). & (5.10)\end{aligned}$$

For $x, y \neq 0$ with the same sign, τ_{x+y} can be computed using the *Kingsbury-Rayner-Formula* [60]:

$$\tau_{x+y} = \tau_y - \lceil S \log_B(1 + B^{(\tau_y - \tau_x)/S})\rfloor. \tag{5.11}$$

If the signs of x and y differ and $\tau_x \neq \tau_y$, we let $z = -|\tau_x - \tau_y|$ and compute

$$\tau_{x-y} = \tau_i - \lceil S \log_B(1 - B^{z/S})\rfloor, \tag{5.12}$$

where $\tau_i := \tau_y$ if $\lambda_1 = 0$ and $\tau_i := \tau_x$, otherwise. The case $\tau_x = \tau_y$ will be handled implicitly in the next section (Secure Operations 5.2.4).

5.2.3.3 Maximum and Minimum

Other operations which are required frequently in arithmetic computations are comparisons. These will be integrated to our framework in form of the functions **LMAX** and **LMIN**, where e.g. **LMAX**$(\overline{x}, \overline{y})$ computes the maximum of $\overline{x}$ and $\overline{y}$. In particular, we need to compare two log-encoded probabilities $\overline{x}$ and $\overline{y}$. This can easily be done by comparing the values τ_x and τ_y, since smaller log-encoded values in fact represent larger values from $\mathbb{R}$. Again we note that a **LMIN** operation can be computed from **LMAX** by swapping the signs σ_x, σ_y of both log-encoded values $\overline{x}, \overline{y}$. To compute the maximum, we use the transformation

$$\mathtt{T}(\overline{z}) = \mathtt{T}(\rho_z, \sigma_z, \tau_z) = \begin{cases} 2^{\delta+1} - \tau_z, & \text{if } \rho_z = \sigma_z = 1 \\ 2^{\delta}, & \text{if } \rho_z = 0 \\ \tau_z, & \text{if } \rho_z = 1, \sigma_z = 0, \end{cases} \tag{5.13}$$

and let $\lambda = 1$ if $\mathtt{T}(\overline{x}) < \mathtt{T}(\overline{y})$ and $\lambda = 0$ otherwise. Now it holds

$$\mathbf{LMAX}(\overline{x}, \overline{y}) = \lambda(\overline{x} - \overline{y}) + \overline{y}.$$

5.2.4 Secure Operations

In this section we describe how to perform secure computations on values that are represented in the encoding described in Section 5.2.1. We consider a two-party scenario, where computation is interactively performed by parties $\mathcal{A}$ and $\mathcal{B}$. Party $\mathcal{A}$ has generated secret keys in the past and is thus the only party who has access to the private key. We encrypt and perform computations with each component of $\overline{x} \in \mathcal{L}_\delta$ separately. While it is possible to use homomorphic encryption to encrypt the flags, we will use garbled circuits to compute with the boolean flags ρ_x and σ_x, whereas the value τ_x will be encrypted using homomorphic encryption. In this setting the keys K_{ρ_x} and K_{σ_x} representing the bits ρ_x and σ_x will be stored by party $\mathcal{A}$. For each arithmetic operation, party $\mathcal{B}$ creates a circuit on demand and sends it to party $\mathcal{A}$. We will denote an encryption of $\overline{x}$ by $\widetilde{x} = (\rho_x, \sigma_x, [\tau_x])$. As homomorphic encryption scheme, we use either the Paillier or DGK cryptosystem (thus, in this section $\langle \cdot \rangle$ refers to either Paillier or DGK).

5.2.4.1 Multiplication and Division

The arithmetic operations **LPROD** and **LDIV** can be computed in a straightforward manner by utilizing the properties of homomorphic encryption (see Section 3.3.1) to implement Equations (5.2) and (5.3). The product operation requires evaluation of two single garbled gates:

$$\begin{aligned}\widetilde{xy} &= \mathbf{LPROD}(\widetilde{x}, \widetilde{y}) \\ &= (\{\rho_x \rho_y\}_{GC}, \{\sigma_x \oplus \sigma_y\}_{GC}, \langle \tau_x \rangle \langle \tau_y \rangle \langle \tau_{C^2} \rangle^{-1}).\end{aligned}$$

For $y \neq 0$ the division can be computed by:

$$\begin{aligned}\widetilde{x/y} &= \mathbf{LDIV}(\widetilde{x}, \widetilde{y}) \\ &= (\rho_x, \{\sigma_x \oplus \sigma_y\}_{GC}, \langle \tau_x \rangle \langle \tau_y \rangle^{-1} \langle \tau_{C^2} \rangle).\end{aligned}$$

Subtraction. As noted above, the operation **LSUB** can be transformed into a sum **LSUM** by evaluating a single garbled gate:

$$\begin{aligned}\widetilde{x - y} &= \mathbf{LSUB}(\widetilde{x}, \widetilde{y}) \\ &= \mathbf{LSUB}((\rho_x, \sigma_x, \langle \tau_x \rangle), (\rho_y, \sigma_y, \langle \tau_y \rangle)) \\ &= \mathbf{LSUM}((\rho_x, \sigma_x, \langle \tau_x \rangle), (\rho_y, \{\neg \sigma_y\}_{GC}, \langle \tau_y \rangle)).\end{aligned}$$

5.2.4.2 Addition

In the following we describe the operation **LSUM** that securely adds two encrypted values $\widetilde{x}$ and $\widetilde{y}$. The flags σ_{x+y} and ρ_{x+y} of the result can be computed in a straightforward manner according to Equations (5.8) and (5.9) using small circuits with 2 and 9 gates each. The values λ_1, λ_2 of Equation 5.4 and 5.5 can be obtained by using standard circuits for comparison and equality testing, while λ_3 and λ_4 can be computed using one garbled gate each.

The main difficulty when computing the sum $\widetilde{x+y}$ of two encrypted values $\widetilde{x}, \widetilde{y}$ consists of computing $\langle \tau_{x+y} \rangle$. This can be achieved by first computing a value $\langle z \rangle = \langle \tau_x - \tau_y \rangle$ and then evaluating the function

$$f_+(z) = \lceil S \log_B(1 + B^{z/S}) \rfloor$$

if $\sigma_x = \sigma_y$, according to Eq. (5.11), and

$$f_-(z) = \lceil S \log_B(1 - B^{-|z|/S}) \rfloor$$

if $\sigma_x \neq \sigma_y$, according to Equation (5.12). Thus, $\langle \tau_{x+y} \rangle$ can be computed once encryptions $\langle \tau_x \rangle$, $\langle \tau_y \rangle$, $\langle f_+(z) \rangle$ and $\langle f_-(z) \rangle$ are available. Since all computations need to be performed in the encrypted domain, we need to evaluate Eq. (5.11) and Eq. (5.12) simultaneously, and later select the correct result. Thus, we compute four boolean flags that select which of the values $\langle \tau_x \rangle$, $\langle \tau_y \rangle$, $\langle f_+(z) \rangle$ and $\langle f_-(z) \rangle$ need to be added to each other. For this reason, we compute

$$\begin{aligned}
\mu_{\tau_x} &= \{\rho_x(\neg\lambda_3 + \lambda_3\lambda_1 + \neg\rho_x)\}_{GC}, \\
\mu_{\tau_y} &= \{\rho_y(\neg\rho_x + \lambda_3\neg\lambda_1)\}_{GC}, \\
\mu_{f_+(z)} &= \{(\neg\lambda_3)\lambda_4\}_{GC} \quad \text{and} \\
\mu_{f_-(z)} &= \{\lambda_3\lambda_4\}_{GC}.
\end{aligned}$$

Note that z takes on values in the interval $\{-2^\delta - 1, \ldots, 2^\delta - 1\}$. The final encryption $\langle \tau_{x+y} \rangle$ can then be computed as

$$\langle \tau_{x+y} \rangle = (\mu_{\tau_x} * \langle \tau_x \rangle)(\mu_{\tau_y} * \langle \tau_y \rangle)(\mu_{f_+(z)} * \langle f_+(z) \rangle)(\mu_{f_-(z)} * \langle f_-(z) \rangle), \tag{5.14}$$

where $*$ denotes a specialized protocol which outputs an encryption of zero if the boolean flag $\mu_i = 0$, and otherwise returns an encryptions of the second operand. The protocol is presented in Section 5.2.5, Protocol 9. For example, for $\rho_x = \rho_y = 1$ and $\sigma_x = \sigma_y = 0$, it holds that $\mu_{\tau_x} = 1$, $\mu_{\tau_y} = 0$, $\mu_{f_+(z)} = 1$ and $\mu_{f_-(z)} = 0$, thus in this case Eq. (5.14) corresponds to Eq. (5.11), addition of two positive values.

Computations of $\langle f_+(z)\rangle$ and $\langle f_-(z)\rangle$ have to be done in a way that no party gains any information about z or $f_\pm(z)$. Since both functions are very difficult to compute analytically on encrypted values, we compute them by running the lookup-table based primitive described in Section 5.1. Thus the accuracy of the secure **LSUM** operation mainly depends on the size δ of the table used in this primitive (see Section 5.2.2).

5.2.4.3 Maximum and Minimum

We construct a basic block $\mathbf{LMAX}(\widetilde{x},\widetilde{y})$ that returns encryptions of the maximum of x and y. For an encrypted log value $\widetilde{z}$, an encryption $\langle \mathtt{T}(\overline{z})\rangle$ of the transformed value of $\overline{z}$ defined in Eq. (5.13) can be computed by

$$\langle \mathtt{T}(\overline{z})\rangle \quad = \quad (\{\rho_z\sigma_z\}_{GC} * (\langle 2^{\delta+1}\rangle\langle -\tau_z\rangle))(\{\rho_z(\neg\sigma_z)\}_{GC} * \langle \tau_z\rangle)(\{\neg\rho_z\}_{GC} * \langle 2^{\delta}\rangle).$$

Let CMP denote a secure comparison protocol as described in Section 3.3.2, i.e. given two tuples with encryptions of $(a_1,b_1),(a_2,b_2)$ it outputs an encryption of the maximum $m \in \{a_1,b_1\}$ along with an encryption of b_m. Then an encryption of the maximum can be computed by

$$\mathbf{LMAX}(\overline{x},\overline{y}) = \mathtt{CMP}((\langle \mathtt{T}(\overline{x})\rangle,\langle\overline{x}\rangle),(\langle \mathtt{T}(\overline{y})\rangle,\langle\overline{y}\rangle)).$$

5.2.4.4 Optimizations

Computations on non-negative values. In case that computations are performed only on positive values (such as probabilities in the interval $[0;1]$), we can further simplify the representation of values, as σ_x will always be set to one and it is not necessary to compute the value f_-. Thus, computing the new sign can be omitted, while computation of the flag ρ and value τ can be simplified.

Computations on strictly positive values. In case that all values are strictly greater than zero the complexity of the proposed protocols decreases even further. We can omit computation of both flags ρ and σ, and the computation of τ_{x+y} can be reduced to $\tau_{x+y} = \tau_x + f_+(z)$. The same optimizations apply, in case that only party $\mathcal{B}$ provides input values in the range $[0;1]$ (and the second party has inputs strictly greater than zero). In this case, party $\mathcal{B}$ can store the flag ρ in plain, omitting the computation of garbled gates and multiplication with τ_x and f_+.

Protocol 9 GC-flag Multiplication protocol.

Input: Party $\mathcal{A}$: ρ_x; Party $\mathcal{B}$: $\rho_0, \rho_1, \langle y \rangle$

Output: Party $\mathcal{B}$: $\langle \rho_x \cdot y \rangle$

```
1: Party $\mathcal{B}$:
   Let $C \in_R \{0,1\}$, $r \in_R \mathbb{Z}_u$
   if $(C == 0)$
      $D = H(K_1), E = H(K_0)$
   else
      $D = H(K_0), E = H(K_1)$
   end
   $\langle y' \rangle = \langle y \rangle^r$
   Send $D, E, \langle y' \rangle$ to $\mathcal{A}$
2: Party $\mathcal{A}$:
   if $H(K_x) == D$
      Send $\langle z_0 \rangle = \langle y' \rangle \langle 0 \rangle$ and $\langle z_1 \rangle = \langle 0 \rangle$ to $\mathcal{B}$
   else
      Send $\langle z_0 \rangle = \langle 0 \rangle$ and $\langle z_1 \rangle = \langle y' \rangle \langle 0 \rangle$ to $\mathcal{B}$
   end
3: Party $\mathcal{B}$:
   $\langle K_x \cdot y \rangle = \langle z_C \rangle^{r^{-1}}$
```

5.2.5 Multiplication protocol

We present a short protocol which allows to multiply a binary value x, held as garbled circuit key K_x by party $\mathcal{A}$ (with $x \in \{0,1\}$), and a homomorphic encryption $\langle y \rangle$, held by party $\mathcal{B}$. During the protocol, party $\mathcal{A}$ stays oblivious about y and the meaning of K_x (i.e. whether $x = 1$ or $x = 0$). Party $\mathcal{B}$ shall not learn the value y, nor which of the two garbled keys for x is held by party $\mathcal{A}$. Protocol 9 depicts the steps performed by party $\mathcal{A}$ and $\mathcal{B}$.

First, party $\mathcal{B}$ uses a hash function $H(\cdot)$ to compute a hash of the two garbled keys for the value x, denoted by K_0 and K_1. Furthermore, in Step 1 we use multiplicative blinding (see Section 3.2.2) to hide the value y. Party $\mathcal{B}$ then sends these values to party $\mathcal{A}$. Note that in it suffices to send only the first bits of D and E, on average this is about two bits each. In Step 2, party $\mathcal{A}$ computes a hash of the value K_x and according to this computes two encryptions z_0, z_1 which she subsequently sends to party $\mathcal{B}$. This allows party $\mathcal{B}$ to compute the final result $\langle K_x \cdot y \rangle$ in Step 3.

In the protocol, two encryptions need to be performed, while the communication complexity consists of 3 encryptions being sent.

5.3 Discussion: Alternative LSUM Implementations

A natural question to ask is, whether **LSUM** can be implemented as efficiently as in Section 5.2.4.2 by using standard SMC methods. In this section we describe three arguments which give rise to the assumption that the function

$$\lceil \log_B(1 + B^{(\tau_x - \tau_y)/S}) \rfloor$$

needed in **LSUM** at a crucial point belongs to the group of functions which can be computed more efficiently by using table look-ups than by arithmetic or binary operations.

- *Computing* $\lceil \log_B(1 + B^{(\tau_x - \tau_y)/S}) \rfloor$ *step by step.* As a first idea, one might try to first compute $y = e^z$ with $z = \tau_x - \tau_y$ and then compute $\log(1 + y)$. Computing $y = e^z$ would be possible for small values of z using the Taylor series of the exponential function. However, the values z appearing in our protocols are typically in the range $[-500, 500]$ which produces results y out of any practical bounds. Furthermore, the implementation of the function $\log(1 + y)$ still seems to be more efficient when done using table look-ups (e.g. see the suggestions for implementing the log function in IEEE 754 arithmetic [61]).

- *Using Taylor series to directly approximate* $\lceil \log_B(1 + B^{(\tau_x - \tau_y)/S}) \rfloor$. Unfortunately, the resulting series shows bad convergence properties. For example, when evaluating the Taylor series at $x_0 = 0$, the series only converges in the interval $[-2; 2]$ with sufficiently good accuracy. Furthermore, securely computing the required powers $(\tau_x - \tau_y)^k$ through secure multiplications is more expensive than doing a single table look-up. The above two arguments combined suggest that a table-lookup is the best suitable salutation for the problem.

- *Implementing the table look-up using garbled circuits.* While a direct implementation using Garbled Circuits and a compiler like Fairplay [44] would have been our first choice, it does not offer any advantages in our context. To perform a table-based **LSUM** operation, we would have to hardcode all table entries in a circuit. In this case, the number of gates to be evaluated just to perform the **LSUM** operations could be roughly estimated by $[\textit{size of tables}] * [\textit{bit length entries}] * [\textit{number of additions}]$, which is beyond practical realization. Thus, we do not expect any improvement when using garbled circuits in a straightforward manner.

Chapter 6

Secure Computations using Floating Point Representation

In this chapter we present an implementation which performs secure computations on values encoded according to the IEEE 754 floating point standard. This answers a long standing open question in the context of secure computations affirmatively: We show that it is indeed possible to perform secure computations on encrypted floating point values. Furthermore, we describe fine-tuned and efficient protocols which operate on such encrypted floating point values.

Reasons to use floating point values over log-encodings as presented in the previous chapter can be manifold. Compared to the approach of Chapter 5, we see various situations where an implementation of the floating point standard can be advantageous: While the table-driven design in Chapter 5 yields to impressive performance when computing with rather small numbers (e.g., with a precision of up to 20 bits), it is clear that for larger values the tables become large and difficult to handle. The tables grow exponentially in the bitlength of the operands, while the implementation in this chapter only has a quadratic complexity. Another advantage of our implementation is a straightforward and easy to implement exception handling. As we will see, we can detect division by zero, over- and underflows at a marginal extra cost in our protocols.

6.1 IEEE 754 Floating Point

We will start by briefly reviewing the IEEE floating point standard [61]. The standard specifies various number formats, operations and exception handling routines. In this chapter we restrict ourselves to describing how to securely represent the binary formats, e.g. floating point values with 32 or 64 bits of total bitlength. Each such value x is stored as a triplet consisting of integers s_x, m_x, e_x such that $x = s_x \cdot m_x \cdot 2^{e_x}$. The first value is the sign, being either -1 or 1. The second value is a significand with either 23 or 52 valid bits. The bitlength of the significand will be denoted as ℓ. Each significand is represented in its *normalized* form, i.e., in the unique binary representation with a leading 1, followed by ℓ significand bits. When storing the significand it suffices to only keep the ℓ remaining bits while the leading 1 can be omitted. The exponent has either 8 or 11 bits, this bitlength will be denoted by ϵ.

The IEEE 754 standard specifies some special values, for which the exponents $e = 0$ and $e = 2^\epsilon - 1$ are reserved. These consist of *Zero*, *Infinity* and *Not A Number.* In the context of SMC, instead of representing these special values explicitly, for efficiency reasons we rather aim at detecting situations where computations yield to a special value. When an arithmetic operation yields an *Infinity* or *Not A Number* as result, an exception is triggered. The standard defines the following exceptions that can occur when handling floating point values:

- Invalid operation (e.g., square root of a negative number),
- Division by zero,
- Overflow (a result is too large to be represented correctly),
- Underflow (result is very small and outside the normal range),
- Inexact (the normalized result of an operation can not be represented exactly, e.g. in case of an underflow).

In this thesis, we limit ourselves to explain how to compute a flag indicating whether such an exception occurred and discuss how to handle it.

6.2 Secure Protocols

When describing the secure protocols, we try to be conform with the floating point standard wherever possible. We will start by giving a protocol for the normalization operation, followed by protocols for the basic arithmetic operations in Sections 6.2.1 to 6.2.4. For a better representation, these protocols do not include handling of exceptions and special values such as *Infinity* or *Not A Number* (see section 6.1). In Section 6.2.5 we explain how to modify the protocols to allow for a basic exception handling.

We use an additively homomorphic encryption scheme to encrypt the values sign s_x, significand m_x and exponent e_x of any floating point value x. To ensure correctness, we encrypt the implicit 1 of each significand along with the ℓ-bit floating point significand. Thus in our implementation, ℓ will be the normal significand length plus one. The special value *Zero* is encoded in a similar way as in the floating point standard, i.e. through $m_x = 0$ and $e_x = 0$. The homomorphic encryption allows to perform arithmetic operations such as addition and subtraction directly on encrypted values. At a key point of our construction we use garbled circuits. Garbled circuits typically require more communication and are less efficient when performing arithmetic operations on large values. However, for operations that need to access single bits, such as in the normalization or integer comparison, they allow for most efficient protocols.

All protocols in this chapter are given in the same setting as in Chapter 5: We assume that all protocols are run by a semi-honest party $\mathcal{B}$, whereas party $\mathcal{A}$ holds the private key for a homomorphic encryption scheme. Party $\mathcal{A}$ only assists in the computations, e.g. when a value needs to be decrypted, or she evaluates a garbled circuit created by party $\mathcal{B}$. In the protocols we use a parameter κ, which can be seen as a statistical security parameter (e.g., choose $\kappa > 80$).

6.2.1 Normalization

Protocol 10 Normalization with extraction of sign

```
Input: $([m], [e])$
Output: $([s_z], [m_z], [e_z])$
1:  Choose $r \in_R \{0, \dots, 2^{\kappa+\mu+1+\epsilon}\}$, $r' \in_R \{0, \dots, 2^{\kappa+\ell}\}$
    $r'' \in_R \{0, \dots, 2^{\kappa+\epsilon}\}$, $r''' \in_R \{0, 1\}$
2:  Compute $[v] = [(m + 2^{\mu}) + 2^{\mu+2}e - r]$,
3:  Construct a garbled circuit $C$ as depicted in Protocol 11
28: Send $C$ and $[v]$ to party $\mathcal{A}$
29: Party $\mathcal{A}$: Decrypt $v$, compute $\tilde{v} = v \bmod 2^{\mu+1+\epsilon}$, evaluate $C$ on $\tilde{v}$
    Send encryptions $[s_{out}], [m_{out}], [e_{out}]$ to party $\mathcal{B}$
30: $[s_z] = [2(s_{out} \oplus r''') - 1]$
31: $[m_z] = [m_{out} - r']$
32: $[e_z] = [e_{out} - r'']$
```

We first describe a secure protocol which allows to normalize values, i.e. which produces a floating point number in the representation according to the IEEE floating point standard. This protocol will be run each time an arithmetic operation occurred. This is necessary because arithmetic operations produce results which do not comply with the standardized representation described in Section 6.2. In particular, the significand of the result of each operation will normally have a bitlength of more than ℓ bits and needs to be adjusted to the exponent.

Protocol 11 Garbled Circuit for signed normalization protocol

```
 4: Input: $\tilde{v}$
 5:     // Remove blinding:
 6:     $v = (\tilde{v} + (r \bmod 2^{\mu+1+\epsilon})) \bmod 2^{\mu+1+\epsilon}$
 7:     $m = v_{\{1,\dots,\mu+1\}}$// Bits 1 to $\mu+1 \to$ significand
 8:     $e = v_{\{\mu+2,\dots,\mu+1+\epsilon\}}$// Remaining Bits $\to$ exponent
 9:     if $(m > 2^{\mu})$ then // Recover sign and abs. value
10:         $m := m \bmod 2^{\mu}$, $\sigma = 1$ // positive sign
11:     else
12:         $\sigma = 0$ // Overflow $\to$ negative
13:         Set bit $z_0 = 0$
14:         for $i = 1$ to $\mu$ // Compute $m := 2^{\mu} - m$
15:             $z_i := \mathrm{OR}(z_{i-1}, m_i)$, $m_i = \mathrm{XOR}(z_{i-1}, m_i)$
16:         end
17:     end
18:     Set $d_{\mu+1} = 0$ // Find leading 1
19:     for $j = \mu$ to 1
20:         $d_j = \mathrm{OR}(m_j, d_{j-1})$
21:     end
22:     Compute $\tilde{e} = \sum_{i=1}^{\mu} d_j$ // Position of leading 1
23:     $s_{out} = r''' \oplus \sigma$
24:     $m_{new} = \sum_{i=1}^{\ell} 2^{i-1} m_{\tilde{e}-\ell+i}$
25:     $m_{out} = r' + m_{new}$
26:     if $(m_{new} == 0)$ then
            $e_{out} = r''$
        else
            $e_{out} = r'' + (e + \tilde{e} - \ell)$
        end
27: Output: $s_{out}, m_{out}, e_{out}$
```

As input the protocol has a significand m and exponent e. We describe two protocols that do the normalization: Protocol 10 takes as input a significand $[m]$ and exponent $[e]$, where the significand can be negative, i.e. $[m]$ contains the sign of the significand. This is the case when running normalization after an addition/subtraction. After a multiplication or division operation, the normalization protocol given in Protocol 12 is run. Here, the significand is always unsigned, thus in this case the protocol is a bit simpler than Protocol 10. Both protocols accept inputs m of bitlength $\mu = 2\ell$ bits, the theoretic maximal deviation from the standard after performing one arithmetic operation, and output normalized significands of size ℓ bits. Since the two protocols are very similar, we limit ourselves to describe only the more complex protocol for normalization of signed values, Protocol 10.

Initially, in lines 1-3, party $\mathcal{B}$ uses additive blinding to hide the input values m and e. Next he creates a garbled circuit as depicted in Protocol 11, and sends both to party $\mathcal{A}$ (line 28). In the garbled circuit, first the blinding factor r is removed (line 6) and the original values m, e are reconstructed (line 7-8). In line 9 we check if $m > 2^{\mu}$. This is the case if and only if the significand was positive before adding 2^{μ} in line 2. Otherwise we compute $2^{\mu} - m$, which in this case corresponds to the absolute value of m.

Lines 18-21 compute a bit array d which has leading zeros followed by bits with value one, where the position of the first bit value one in d coincides with the leading one in m. Next, we compute a value $\tilde{e} = \sum_{i=1}^{\mu} d_j$. More efficient than using μ binary adders is to use a multiplexer construction. Such a construction is able to find the first value d_i with $d_i = 1$ with at most μ multiplexers. For simplicity of notation we use the additive notation. The final output bits of $\sum_{i=1}^{\ell} 2^{i-1} m_{\tilde{e}-\ell+i}$ can again be computed most efficiently using ℓ μ-1-Multiplexers. Note that for bit values m_i, where $i \leq 0$, a value $m_i = 0$ will be inserted. This is the case if the significand is small (e.g. after a Subtraction), and needs to be shifted to the left.

Finally, in lines 30-32, party $\mathcal{B}$ removes the blinding factors and computes the final result. Note that we use $\oplus$ to denote the bitwise XOR operation, thus $2(s_{out} \oplus r''') - 1$ maps the binary value $s_{out} \oplus r'''$ to $\{-1, 1\}$.

Complexity. During the protocol 7 values need to be encrypted, while one value is decrypted. The garbled circuit has approximately $2\kappa + (\ell + 6)\mu + 3\epsilon$ gates, e.g. for $\ell = 24$, $\mu = 48$, $\epsilon = 8$ and $\kappa = 80$ this results in a circuit with approximately 1650 gates.

6.2.2 Subtraction and Addition

Subtraction and addition of floating point values $([s_x], [m_x], [e_x])$, $([s_y], [m_y], [e_y])$ are rather complex operations. Protocol 13 depicts the secure operations performed by party $\mathcal{B}$ in order to add or subtract encrypted values to obtain the result

Protocol 12 Normalization for unsigned significands

```
Input: $([m], [e])$
Output: $([m_z], [e_z])$
 1: Choose $r \in_R \{0, \ldots, 2^{\kappa+\mu+\epsilon}\}$, $r' \in_R \{0, \ldots, 2^{\kappa+\ell}\}$, $r'' \in_R \{0, \ldots, 2^{\kappa+\epsilon}\}$
 2: Compute $[v] = [m + 2^{\mu+1}e - r]$,
 3: Construct a garbled circuit $C$ which realizes the functionality as follows:
     4: Input: $\tilde{v}$
     5:     // Remove blinding:
     6:     $v = (\tilde{v} + (r \bmod 2^{\mu+\epsilon})) \bmod 2^{\mu+\epsilon}$
     7:     $m = v_{\{1,\ldots,\mu\}}$// Bits 1 to $\mu \rightarrow$ significand
     8:     $e = v_{\{\mu+1,\ldots,\mu+\epsilon\}}$// Remaining Bits $\rightarrow$ exponent
     9:     Set $d_{\mu+1} = 0$ // Find leading 1
    10:     for $j = \mu$ to 1
    11:         $d_j = \mathrm{OR}(m_j, d_{j-1})$
    12:     end
    13:     Compute $\tilde{e} = \sum_{i=1}^{\mu} d_j$ // Position of leading 1
    14:     $m_{new} = \sum_{i=1}^{\ell} 2^{i-1} m_{\tilde{e}-\ell+i}$
    15:     $m_{out} = r' + m_{new}$
    16:     if $(m_{new} == 0)$ then
    17:         $e_{out} = r''$
    18:     else
    19:         $e_{out} = r'' + (e + \tilde{e} - \ell)$
    20:     end
    21: Output: $m_{out}, e_{out}$
22: Send $C$ and $[v]$ to party $\mathcal{A}$
23: Party $\mathcal{A}$: Decrypt $v$, compute $\tilde{v} = v \bmod 2^{\mu+1+\epsilon}$ and evaluate $C$ on $\tilde{v}$
24: Send encryptions $[m_{out}], [e_{out}]$ to party $\mathcal{B}$
25: $[m_z] = [m_{out} - r']$
26: $[e_z] = [e_{out} - r'']$
```

$([s_z], [m_z], [e_z])$. For this, the sign value is multiplied to the significand (line 28). Before we can add the two significands, they need to be adjusted to the same exponent (lines 26-28). For this, we first select the value m_{max} that belongs to the larger exponent. This can be done using a value λ with $\lambda = 1$ if $e_x < e_y$ and $\lambda = 0$ otherwise. Next, we compute a shift value *shift*, such that *shift* $= [2^{\mathrm{abs}(e_x - e_y)}]$ if $\mathrm{abs}(e_x - e_y) < \ell$, and *shift* $= 0$ otherwise. This shift value is multiplied to the significand that belongs to the smaller exponent. In order to determine the value *shift* as well as the value λ, indicating the larger exponent, a garbled circuit as

Protocol 13 Addition and Subtraction

Input: $([s_x],[m_x],[e_x]),([s_y],[m_y],[e_y])$

Output: $([s_z],[m_z],[e_z])$

```
1: Choose $r \in_R \{0,\ldots,2^{2^{\kappa+2\epsilon}}\}$, $r' \in_R \{0,\ldots,2^{2^{\kappa+\phi}}\}$, $r'', r''' \in_R \{0,1\}$
2: $[v] = [e_x + 2^\epsilon e_y - r]$
3: Construct a garbled circuit $C$ as depicted in Protocol 14
24: Send $C$, $[v]$ to party $\mathcal{A}$
25: Party $\mathcal{A}$: Decrypt $[v]$, compute $\tilde{v} = v \bmod 2^{2\epsilon}$, evaluate $C$ and send encryp-
    tions $[2^o], [p], [q]$ to party $\mathcal{B}$
26: $[shift] = [2^{diff}] = [2^o \cdot 2^{-r'}] \cdot [-p \oplus r'']$ // $shift = 2^{o-r'} - z$
27: $[\lambda] = [q \oplus r''']$ // Recover $\lambda$
28: $[m'_x] = [s_x] * [m_x]$, $[m'_y] = [s_y] * [m_y]$
29:  $[m_{max}] = ([m'_x] * [\lambda]) \cdot ([m'_y] * [1-\lambda])$
30: $[m_{min}] = [m'_x + m'_y - m_{max}]$
31:  $[m'] = [m_{max}] \cdot ([m_{min}] * [shift])$
32:  $[e'] = ([e_x] * [\lambda]) \cdot ([e_y] * [1-\lambda])$
33:  $([s_z],[m_z],[e_z]) = \text{Normalize}([m'],[e'])$
```

depicted in Protocol 14 is used. The circuit first recomputes the two exponents e_x, e_y, which have been additively blinded by party $\mathcal{B}$ (see lines 1,2, 5-8). Next, the value λ is computed. This can be done using a comparison circuit with at most ϵ gates [29]. Then, the absolute difference $diff = e_x - e_y$ is analyzed. In case that $diff > \ell$, a value $diff = 0$ should be returned. In lines 10 to 12, we first check if there is a bit difference in the leading $\epsilon - \phi$ bits, with $\phi = \lceil \log_2(\ell) \rceil$. In this case, the absolute distance is greater or equal than $2^\phi > \ell$. Next, it suffices to analyze the remaining ϕ bits. In lines 14 to 16 we compute the correct absolute difference $e = \text{abs}(a-b) = \text{abs}(e_x - e_y)$, in case that the leading $\epsilon - \phi$ bits are all equal. This value is compared to ℓ in line 17. As final result, the circuit outputs blinded values *diff* and z, for which it holds that $2^{diff} - z = shift$. Note that, since the value $o = diff{+}r$ is in the exponent of 2^o, a value $shift = 0$ can not be recovered directly when removing the blinding factor (e.g. $2^r \cdot 2^{-r} = 1$). For this reason, the value $z = 1$ is subtracted in case that *shift* should be 0.

Finally, a normalization is triggered. Since the sign of the result is still contained in the value m', it is necessary that during normalization this value is extracted and returned as encryption $[s_z]$.

Protocol 14 Garbled Circuit for Addition/Subtraction Protocol

```
Garbled Circuit C
 4:   **Input:** $\tilde{v}$
 5:   // Remove blinding:
 6:   $v = (\tilde{v} + (r \bmod 2^{2\epsilon})) \bmod 2^{2\epsilon}$
 7:   $a = v_{\{1,\dots,\epsilon\}}$// Bits 1 to $\epsilon \to e_x$
 8:   $b = v_{\{\epsilon+1,\dots,2\epsilon\}}$// Bits $\epsilon + 1$ to $2\epsilon \to e_y$
 9:   if $(a < b)$ then $\lambda = 1$ else $\lambda = 0$
10:   for $i = \phi + 1$ to $\epsilon$ // Check MSBs
11:     $s_i = \mathrm{XOR}(a_i, b_i)$, $t_i = \mathrm{OR}(s_i, t_{i-1})$
12:   end
13:   $u := t_\epsilon$ // if $u == 1$, then $\mathrm{abs}(a - b) > \ell$
14:   $a := a \bmod 2^\phi$, $b := b \bmod 2^\phi$
15:   $c = a - b$, $d = b - a$
16:   if $(c < 0)$ then $e = d$ else $e = c$
17:   if $(e > \ell)$ then $u := 1$ else $u := 0$
18:   if $(u == 0)$ then
19:     $diff = e$, $z = 0$
20:   else
21:     $diff = 0$, $z = 1$
22:   end
23: Output: $o = diff + r'$, $p = \mathrm{XOR}(z, r'')$, $q = \mathrm{XOR}(\lambda, r''')$
```

Complexity. In Protocol 13, 7 secure multiplications along with 5 encryptions and one decryption are performed in the homomorphic encryption scheme. The garbled circuit has $\kappa + 4\epsilon + 5\phi$ gates. For 32-bit floating point values, with $\kappa = 80$, $\epsilon = 8$ and $\phi = 5$ this results in a circuit with roughly 200 gates. Furthermore, we have to add the complexity for the normalization (Protocol 10).

6.2.3 Multiplication

Multiplication (Protocol 15) of two encrypted floating point values can be reduced to secure operations on homomorphic encryptions. Both sign values and significands can be multiplied, requiring one secure multiplication each, while the exponent of the result is the sum of e_x and e_y. Finally a normalization is required. Since the sign can be computed separately, the normalization protocol only has to adjust significand and exponent.

Protocol 15 Multiplication

```
Input: $([s_x], [m_x], [e_x]), ([s_y], [m_y], [e_y])$
Output: $([s_z], [m_z], [e_z])$ where $z = x \cdot y$
1: $[s_z] = [s_x] * [s_y]$
2: $[m'_z] = [m_x] * [m_y]$
3: $[e'_z] = [e_x] \cdot [e_y]$
4: $([m_z], [e_z]) = \text{Normalize}([m'_z], [e'_z])$
```

Complexity. In the protocol, only 2 secure multiplications and a normalization have to be performed.

6.2.4 Division

The secure protocol for division is depicted in Protocol 16. Initially, party $\mathcal{B}$ additively blinds the two significands m_x and m_y, prepares a garbled circuit C and both C and the value v them to party $\mathcal{A}$. The garbled circuit C implements a standard pencil and paper division method: First, in lines 5-7 the two operands are recovered. Next, in line 8 values p and d are initialized with the bits $v_1, \ldots, v_\ell$ and $v_{\ell+1}, \ldots, v_{2\ell}$ according to operands m_x and m_y. The major steps of the division algorithm are contained in lines 10 and 11: Line 10 implements a left shift, followed by a trial subtraction in line 11. In line 12 the quotient bit is written accordingly.

Complexity. During the protocol 3 values need to be encrypted and 1 value decrypted. The circuit has approximately $\kappa + 2\ell^2 + 5\ell$ gates. For $\ell = 24$ this results in about 1400 gates. In addition, the complexity of the normalization has to be added.

6.2.5 Exception Handling

In this section we describe how exceptions can be detected and suggest strategies on how to treat them.

6.2.5.1 Detection

In order to detect whether an exception has occured, we suggest to introduce a new flag E, which is kept as encrypted value by party $\mathcal{B}$. This flag will be added as input to all secure protocols, and an updated version will be returned as

Protocol 16 Division

```
Input: Party $B$: $([s_x], ([m_x], [e_x]), (([s_y], [m_y], [e_y])$
Output: Party $B$: $(([s_z], [m_z], [e_z])$
1: Choose random $r' \in_R \{0, \ldots, 2^{2\ell+\kappa}\}$, $r \in_R \{0, \ldots, 2^{\ell+\kappa}\}$
2: Compute $[v] = [m_x + 2^\ell m_y - r']$
3: Construct a garbled circuit $C$ which realizes the functionality as follows:
     Garbled Circuit C
     4: Input: $\tilde{v}$
     5:     $v := (\tilde{v} + (r' \bmod 2^{2\ell})) \bmod 2^{2\ell}$
     6:     $a = v_{\{1,\ldots,\ell\}}$
     7:     $b = v_{\{\ell+1,\ldots,2\ell\}}$
     8:     Set $p_i = a_i$, $d_{i+1} = b_i$ for $i \in \{1, \ldots, \ell\}$
     9:     for $i = \ell + 2$ to 1
     10:        $p_{\ell+2} := p_{\ell+1}, \ldots, p_2 := p_1, p_1 := 0$
     11:        $p' := p - d$
     12:        if $(p' < 0)$ then
                    $q_i = 0$
     13:        else
                    $p := p'$, $q_i = 1$
                end
            end
     14:    $o = q + r$
     15: Output: $o$
   Send $C$, $[v]$ to party $\mathcal{A}$
4: Party $\mathcal{A}$: Evaluate $C$ on $\tilde{v} = v \mod 2^{2\ell}$, obtain $o$, send $[o]$ to Party $\mathcal{B}$
5: Run Normalization on $([o - r], [e_y - e_x])$
```

output of each protocol. Alternatively, if a more fine grained exception handling is necessary, one may want to keep five different flags, one for each type of exception. We discuss in short the five types of exceptions and explain how to modify the protocols to allow for their detection.

The *Division by zero*-exception can only be triggered in Protocol 16. When creating the garbled circuit, party $\mathcal{B}$ adds the exception flag E as input and output to the circuit. The divisor is equal to 0 if and only if $b_\ell = 0$ (the most significant bit) in line 7. Thus, the value $\mathrm{OR}(E, 1 - b_\ell)$ is computed and output as the new exception flag.

The *Overflow/Underflow*-exception is raised whenever a value is to large or to small to be represented in the given number format. Since we use a homomorphic encryption scheme with sufficiently large plaintext space (e.g. for Paillier the bitlength k of the plaintext space is typicall much larger than ℓ), it suffices to check for an overflow only during normalization. Here an overflow/underflow can be detected when the exponent is adjusted in the last line of Protocol 10. The simplest way of doing this is by running a state of the art comparison protocol in the garbled circuit in Protocol 10 and compute the exception flag as $E := (e_{out} \geq 2^{\epsilon})$.

The exceptions *Inexact* and *Invalid Operation* are typically triggered jointly with or after one of the other exceptions have been triggered. In this case the flag E has been already set. Other reasons for which these types of exceptions can be triggered depend on the application, e.g. signalize a negative input to a square root operation and are therefore out of scope for this investigation.

6.2.5.2 Strategies for Exception Handling

We suggest three strategies for exception handling. In the context of secure two party computation, exceptions have to be handled particularly careful, since revealing that an exception has occurred can reveal much information about the private inputs of the participating parties. However, any exception naturally renders all subsequent computations invalid, and thus should be detected. Therefore, for each application and scenario the strategy for exception handling should be examined and reconsidered. Three basic strategies can be envisioned:

- *Strategy 1:* After each arithmetic operation, one checks whether an exception occurred. On the one hand, this strategy allows maximal transparency, and allows the two parties to detect an exception as early as it occurred. On the other hand, this strategy allows to draw conclusions about the input values of each party, which might have led to a certain exception.
- *Strategy 2:* Periodically, e.g. after k arithmetic operations, one decrypts and checks the exception flag. This strategy is similar to Strategy 1. However, it provides some more privacy since it is not immediately clear when the exception occurred.
- *Strategy 3:* Arrange all arithmetic computations in a way that, given that the semi-honest parties provide correct inputs, no exceptions can occur. In this case it suffices to check the exception flag only for the final result or even omit its computation at all.

There is no general rule to tell which of the proposed strategies can be seen as the best. Clearly, the first strategy allows to detect exceptions as early as they occur. However, this strategy also reveals more information than the other strategies. For example, in case that the first operation is a division operation computed on secret inputs, triggering an division by zero exception immediately reveals that one of the secret inputs was equal to zero. Thus, when implementing the protocols one has to decide which strategy to choose depending on the application scenario.

6.3 Security

We sketch a security proof in the semi-honest attacker model. We need to prove that the views of both parties can be efficiently simulated (i.e. their simulated views are statistically indistinguishable from a view of a real protocol run, see [10] for details). Security against party $\mathcal{B}$, in the asymmetric two party scenario, follows directly from the composition theorem for the semi-honest model (Theorem 7.3.3 in [10]), the security proof for garbled circuits [62] and the fact that $\mathcal{B}$ only sees semantically secure encryptions, which can be simulated by randomized encryptions of $[0]$. Security against party $\mathcal{A}$: Whenever party $\mathcal{A}$ is to decrypt some value $[x + r]$, the simulator sends an encryption of some value r', chosen uniformly at random from the same range as the blinding factor r used in the protocol. This allows for a simulator proof, using the same arguments as for party $\mathcal{B}$.

Chapter 7

Discussion

In this chapter we discuss the properties of the two approaches to securely compute with non-integer values presented in Chapters 5 and 6, and compare it to the basic solution of Chapter 4. We will highlight their main differences and indicate for which kind of application a specific approach might be advantageous. The main part of this chapter will consist of an complexity analysis, comparing the schemes presented in Chapters 5 and 6 with an alternate fixed point representation with sufficiently high precision.

7.1 Basic Properties

Given a fixed precision of ℓ bits, both schemes, the logarithmic encoding in Chapter 5 and the floating point representation in Chapter 6 can approximately represent 2^ℓ different values. Both schemes are flexible in the sense that the distance between largest and smallest represented number can be chosen by adjusting parameters. For the logarithmic encoding the relative error is constant. In a floating point representation the relative representation error is also strictly bounded. Thus, the main differences of the schemes are in their efficiency, as will be seen in the next section.

7.2 Complexity Analysis

In this section we will describe a theoretical analysis of the complexity of the two approaches presented in Chapters 5 and 6. The complexity will then be compared with an alternate fixed point representation with sufficiently high precision (see Chapter 4). For the comparison, we will assume that the probabilistic rounding method (Protocol 3) is used and at most one multiplication is performed. This approach serves well for a comparison, due to the fact that the complexity of the probabilistic rounding protocol is easy to analyze and to the fact that this protocol yields to most efficient schemes for computations in fixed point representation [49]. In case that more consecutive multiplications need to be performed or a different rounding protocol will be used performance will deteriorate; thus the results presented in this chapter can be seen as optimistic lower bounds for the complexity of a SMC fixed point representation scheme.

Again, we assume that ℓ bits are used to represent a value in the logarithmic encoding or the floating point representation.

The cost for performing arithmetic operations is the main difference that distinguishes the approach using log encoding from the floating point representation. While the protocols for all arithmetic operations have about the same complexity for floating point values, it can be seen that the situation is inherently different in the setting where log encodings are used. In this setting, no efficient method for computing the **LSUM** operation is known, i.e. one either has to compute logarithms under encryption analytically, or (as done in Chapter 5), a solution based on table-lookups has to be used. The latter allows for efficient solutions for small values ℓ; however, the complexity of the **LSUM** operation grows exponential in the parameter ℓ. As illustrated in Section 5.3, an analytic computation of the **LSUM** seems impractical for all values ℓ. While on the one hand additions and subtractions are rather expensive operations, on the other hand multiplications, divisions and even exponentiations can be performed very efficiently on log-encoded values.

7.2.1 Computational Complexity

Table 7.1 depicts the computational complexity of the three approaches: The first column depicts the computational complexity of arithmetic protocols in log encoding, the second for floating point and the third column for the basic implementation using a fixed point representation (see Section 4).

As a complexity measure we count the number of 1024-bit modulo multiplications. In order to express one multiplication of values with 1024 bits bitlength, we assume that the computational complexity of one multiplication grows quadratic with the bitlength of the operands. We further assume that one modular exponentiation requires $1.5k$ modular multiplications, where k is the bitlength of the exponent. This is a realistic assumption, since on average about 50% of the bits in the exponent will be equal to zero. In the complexity evaluation we neglect the computational cost for evaluating garbled circuits, i.e., we assume that a call to a hash function is negligible compared to one modular multiplication. This makes the complexity of the floating point implementation appear to be constant; however, for the typical parameter sizes used in the experiments this is a realistic assumptions, since the circuits are rather small. For larger parameters ℓ one should also consider the complexity of garbled circuits in the floating point approach. The complexity of operations using logarithmic encoding and floating point representation were derived by counting the number of operations that occur during each protocol. For the log-encoding we assumed a fixed set of parameters $\kappa = 50, t = 256$, $C = 1$, $S = 10$ and $B = 2$. The parameters κ and t would normally be adjusted to the length of the parameter ℓ (to improve performance), for simplicity and to allow for a direct comparison we assume those parameters are fixed. Parameter $C = 1$ yields to computations on values with absolute value less than 1, this corresponds to the estimations we made for the fixed point representation where we also only consider the fractional part for the complexity analysis.

Protocol	Log-Encoding	Floating Point	Fixed Point
Sum/Subtract	$946\ell + 8,709$	230,532	$2^{2\ell-23}$
Product	2	85,916	$2^{3\ell-22}$
Division	2	52,340	—

Table 7.1: Computational Complexity Measure for Secure Protocols.

The complexity of the fixed point representation can be estimated as follows: Assume that we use a log encoding with ℓ bit operands. In order to represent small values with the same accuracy, a fixed point representation scheme should be able to represent the two smallest numbers n_1, n_2 that can be represented by the log encoding. This is the case if the absolute error of the fixed point representation is smaller than $(n_1 - n_2)/2$. For a fixed point representation scheme which uses ℓ_{FP} bits for the fractional part, the absolute representation error is $2^{-\ell_{FP}-1}$. Thus it holds

$$\begin{array}{rrcl}
 & 2^{-\ell_{FP}-1} & \overset{!}{<} & (n_1 - n_2)/2 \\
\Leftrightarrow & 2^{-\ell_{FP}} & < & (n_1 - n_2) \\
\Leftrightarrow & 2^{-\ell_{FP}} & < & C \cdot B^{-(2^\ell-2)/S} - C \cdot B^{-(2^\ell-1)/S} \\
\Leftrightarrow & -\ell_{FP} & < & \log_2(C) + \log_2(B^{(2-2^\ell)/S} - B^{(1-2^\ell)/S}) \\
\Leftrightarrow & -\ell_{FP} & < & \log_2(B^{(1-2^\ell)/S}) + \log_2(B^{1/S} - 1) \\
\Leftrightarrow & -\ell_{FP} & < & (1 - 2^\ell)/10 - 3.8 \\
\Leftrightarrow & \ell_{FP} & > & 0.1 \cdot 2^\ell + 3.7.
\end{array} \tag{7.1}$$

In order to be able to compute the product of two values stored in fixed point representation, the plaintext space of the cryptoosystem has to be twice as large as $0.1 \cdot 2^\ell + 3.7$. This is necessary since temporary results occur which have twice the bitlength of a normal value stored in fixed point representation. Thus we set $\ell_{FP} := 0.1 \cdot 2^{\ell+1} + 7.4$ and use this parameter as minimal bitlength of the plaintext space. We will assume that the values in fixed point notation will be encrypted using Paillier encryption.[1] Assuming that the complexity of one modular multiplication grows quadratic in the operands size, it holds that a multiplication of two ciphertexts of size $2\ell_{FP}$ is about $(\frac{2\ell_{FP}}{1024})^2$ more expensive than a 1024-bit multiplication. As each encryption in the Paillier cryptosystem requires one exponentiation of an exponent of size ℓ_{FP}, it can be seen that the complexity of one Paillier encryption grows with ℓ_{FP}^3, i.e. cubic complexity in ℓ_{FP}. Furthermore, according to Inequality (7.1), the parameter ℓ_{FP} has an exponential dependence on the parameter ℓ. Using the above estimations and counting the number of operations required for each arithmetic operation yields to the lower bounds for the computational complexity depicted in the third row of Table 7.1. In particular, Table 7.1 shows that for larger parameters of ℓ the fixed point representation can not be implemented efficiently. In particular, even for small parameters ℓ it can not compete with log-encodings in terms of computational complexity if an equal amount of additions and multiplications should be performed.

We note that, in fixed point representation, the complexity of the division operation heavily depends on the algorithm which is used for the division and is therefore omitted in this presentation.

Figures 7.1 and 7.2 plot the computational complexity of one addition (Figure

[1]Naturally, one could as well use Damgård-Jurik cryptosystem [19], which would be slightly more efficient. However, this would complicate the complexity analysis even though the asymptotic complexity is the same as for Paillier.

7.1) and one multiplication (Figure 7.2) in each of the different representation schemes. The Y-axis plots the number of 1024-bit multiplications, including the hidden constants which have been omitted in Table 7.1 and under the assumption that the minimal length of the Paillier plaintext space is 1024 bits.

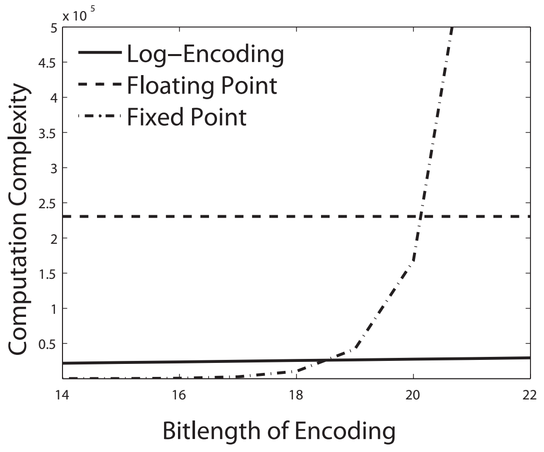

Figure 7.1: Computational Complexity of Addition/Subtraction.

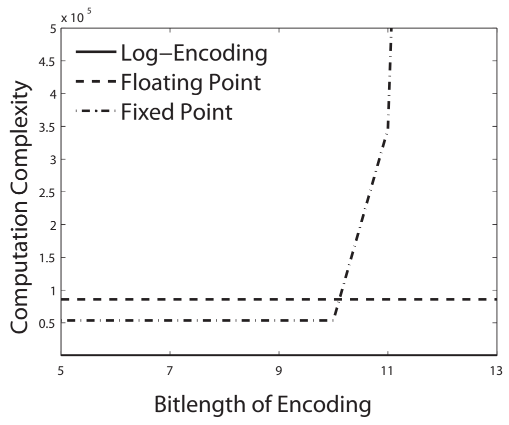

Figure 7.2: Computational Complexity of one Multiplication.

7.2.2 Communication Complexity

Table 7.2 depicts the communication complexity of the secure protocols for arithmetic computations. Again, the complexity is presented as a function of the parameter ℓ, all values are given as kilobytes. As it can be seen, product and division operations for log encodings as well as additions and subtractions for the fixed point representation can be performed at essentially no cost. However, both

Protocol	Log-Encoding	Floating Point	Fixed Point
Sum/Subtract	$\ell 2^{\ell-12}$	$0.02\ell^2 + 0.22\ell + 14.28$	0
Product	0.03	$0.02\ell^2 + 0.15\ell + 7.19$	$0.1 \cdot 2^{\ell-11}$
Division	0.03	$0.05\ell^2 + 0.26\ell + 8.53$	—

Table 7.2: Communication Complexity for Secure Protocols in kilobytes.

schemes exhibit an exponential increase in complexity for the remaining operations. In particular, the communication complexity of the **LSUM** operation is dominated by the term $\ell 2^{\ell-12}$ which corresponds to the size of the table transferred during the table look-up. For the fixed point representation, the plaintext space grows exponentially fast with the parameter ℓ (see Section 7.2.1). In order to show the effect of hidden constants (which have been omitted in Table 7.2) for small values ℓ we present Table 7.3, which depicts the overall communication for one protocol run for a given parameter ℓ. It can be seen that for parameters $\ell \leq 15$ the fixed point representation has acceptable communication complexity, while the floating point representation is preferable for larger parameters ℓ.

We plotted the communication complexity of one addition in Figure 7.3 and of one multiplication in Figure 7.4. The Y-axis plots the communication complexity in kilobytes. Again, the plots include the hidden constants which have been omitted in Table 7.2 and made the assumption that the minimal length of the Paillier encryption is 2048 bits.

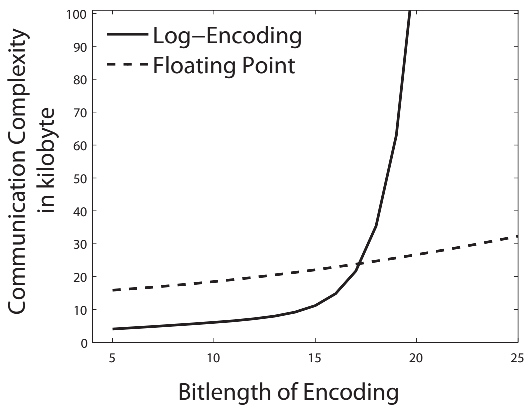

Figure 7.3: Communication Complexity of Addition/Subtraction.

	Addition/Subtraction			Multiplication		
ℓ	Log	Float	Fixed	Log	Float	Fixed
5	4,09	15,88	0,00	0,03	8,44	1,25
6	4,46	16,32	0,00	0,03	8,81	1,25
7	4,85	16,80	0,00	0,03	9,22	1,25
8	5,25	17,32	0,00	0,03	9,67	1,25
9	5,66	17,88	0,00	0,03	10,16	1,25
10	6,10	18,48	0,00	0,03	10,69	1,25
11	6,60	19,12	0,00	0,03	11,26	1,25
12	7,20	19,80	0,00	0,03	11,87	1,25
13	8,00	20,52	0,00	0,03	12,52	2,01
14	9,21	21,28	0,00	0,03	13,21	4,01
15	11,20	22,08	0,00	0,03	13,94	8,01
16	14,82	22,92	0,00	0,03	14,71	16,01
17	21,69	23,80	0,00	0,03	15,52	32,01
18	35,41	24,72	0,00	0,03	16,37	64,01
19	62,98	25,68	0,00	0,03	17,26	128,01
20	119,75	26,68	0,00	0,03	18,19	256,01
21	236,23	27,72	0,00	0,03	19,16	512,01
22	478,25	28,80	0,00	0,03	20,17	1024,01
23	976,31	29,92	0,00	0,03	21,22	2048,01
24	2014,37	31,08	0,00	0,03	22,31	4096,01
25	4160,43	32,28	0,00	0,03	23,44	8192,01

Table 7.3: Communication in kilobytes for selected parameters ℓ.

7.2.3 Overall Comparison

While it is difficult to highlight one of the presented approaches as the best for arbitrary parameters ℓ, it is possible to give guidelines on which computational scheme should be used: Unsurprisingly, for small parameters ℓ the fixed point implementation yields to good performance for all arithmetic operations. However, for parameters $\ell \geq 11$ the fixed point representation becomes computationally to complex. Depending on the application, even for smaller parameters we suggest to use the log encoding, as for this scheme the computational complexity on average[2] is lower than for any of the others, and the communication complexity

[2]We assume that an equal amount of additions and multiplications has to be performed.

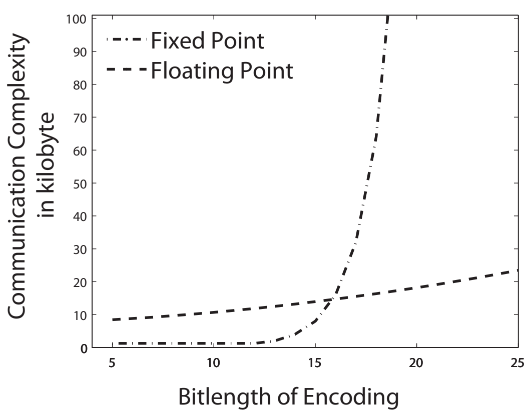

Figure 7.4: Communication Complexity of one Multiplication.

of the log encoding scheme is still better than for the floating point representation. For much larger parameters, the floating point representation scheme will be preferable over both fixed point and log encoding. The break even point for the floating point scheme depends on whether communication or computation is more costly. For current computation- and network hardware, we estimate this parameter to be in the range between $\ell = 18$ and $\ell = 20$. Clearly, for a given setup this threshold can easily be determined experimentally.

The choice of the representation scheme will also depend on the type of application that should be implemented. For example, for an application which mainly consists of products/divisions it might be still more efficient to implement the log encoding even for large parameters ℓ, even though the asymptotic complexity of the floating point scheme seems to be lower. In particular if an exponentiation with known exponent should be performed as a secure computation, even for small ℓ the log encoding might be advantageous even over a fixed point representation.

Chapter 8

Applications

As an application we describe in this chapter how the computational framework presented in Chapter 5 can be applied to algorithms that securely analyze genomic and gene product related sequences by using Hidden Markov Models (HMM). In particular, we consider a scenario as introduced in Chapter 1. Two parties jointly want to evaluate a Hidden Markov Model (HMM) on a given DNA or protein sequence. Party $\mathcal{A}$, who is a health care provider acting on behalf of a patient, has sequenced the patient's genome. $\mathcal{A}$ wants to interact with a provider $\mathcal{B}$, who offers a service to check parts of the genome against a model, encoded as HMM, for a specific disease. The information "how good" the model fits the genome determines the likelihood of a disease predisposition, and is therefore interesting to know for $\mathcal{A}$ and his patients. However, party $\mathcal{A}$ is bound to preserve the patient's privacy, since DNA data is considered highly sensitve private information. At the same time, party $\mathcal{B}$ does not want to disclose the HMM since the model itself is his business secret that distinguishes his service from other providers.

In this chapter we show how $\mathcal{A}$ and $\mathcal{B}$ can achieve both goals by providing a way to run the *forward algorithm* on the sequence and HMM in an oblivious manner. As a second application, we consider the case, where both parties want to run a secure version of the *Viterbi algorithm*. In this setting, the result of the Viterbi algorithm explains how the genome sequence can be generated by the model.

8.1 Secure Bioinformatics and HMM Evaluation

In this section we review related work in the field of secure bioinformatics. Some works deal with the secure analysis of genomic sequences by providing secure string processing algorithms: [63] considers the problem of securely computing the edit distance between two strings; [64] presents a secure algorithm to perform sequence alignments using the Smith-Waterman algorithm; [65] considers the problem of securing distributed computations on genomic sequences; [58] proposes an algorithm that allows to run arbitrary queries, formulated as regular expressions, on genomic sequences in an oblivious way; finally, [66] shows how to find a certain pattern in a DNA sequence. These existing works considered basic string-based bioinformatics algorithms. In contrast we show how to evaluate much more complex probabilistic queries represented by Hidden Markov Models (HMMs) on genomic data in a secure way. HMMs are widely used in computational biology to detect locations with certain properties, e.g., disease related signals [67, 68, 69] or general properties of important, disease related enzymes, such as kinases [70, 71, 72].

Some authors considered the problem of securely evaluating HMMs: [73] contains an allegedly secure way of evaluating Hidden Markov Models on strings. However, their construction, which uses secret sharing, contains several inaccuracies. First, they base their protocols on a cryptographic building block which computes the index of the maximum of n shared numbers, which leaks information and is thus not provably secure; replacing this primitive in [73] with a secure one is possible. However this results in a severe performance penalty.

Second, and more importantly, their way of computing sums of values represented in logarithmic notation tends to get numerically unstable: they compute with cryptographic shares of numbers as if they were reals (in particular, raise the real number e to the power of large, cryptographic shares and take the natural logarithm), and treat the results as cryptographic shares as well, without considering severe rounding errors, which will propagate through the entire computation. Unfortunately, [73] does not present any experimental results to quantify this error.

[74] gives another solution to securely evaluating HMMs, where a third party in form of a commodity server is involved in the computations. This party necessarily needs to be independent and is not allowed to collude with any of the parties, otherwise all privacy of the other party will be lost (and, maybe even worse, the colluding parties can manipulate the result of the computations at

their will). Even without collusion, it is not fully clear if the protocols compute correct results, and compute them in a secure way. This is due to the fact that in the protocols some additive blinding is used to hide temporary results. However, it is never mentioned if the computations are performed in a finite field, the integers or real numbers. This in turn makes it hard to estimate from which domain the blinding factors are chosen and to evaluate the overall security of the protocol. Similar to [73], there are no sufficient experimental results to tell whether the protocols would really provide accurate results for HMMs of realistic size. In fact, the experiments were only run for sub-protocols on randomly chosen inputs and were only analyzed due to timing aspects; the algorithm has not been tested on real-world HMMs.

8.2 Hidden Markov Models

Hidden Markov Models (HMM) are probabilistic methods that model a stream of symbols by a Markov process, which is driven by “hidden” states of the process, unobservable to an outsider. In different states the dynamics of the process differs and thus the statistics of emitted symbols varies with time. States in bioinformatics applications can be indicators for particular disease related properties of sequences that code for e.g. non-beneficially mutated proteins or sites of potential post-translationally modifications. HMMs are not only used in bioinformatics, but have a wide range of applications such as speech recognition [75], pattern recognition [76] or image classification [77].

More formally, a Hidden Markov Model $\lambda = (A, B, \pi)$ is characterized by the following elements:

- A set of $S = \{S_1, S_2, \dots, S_N\}$ of N “sites” (hidden states), which reflects experimental observations such as single nucleotides or other more general symbols. Any sequence of length T, $q_1 q_2 \dots q_T$, of states is called a path Q through the sites.

- A set V of M distinct observation symbols per state (the output alphabet): $V = \{v_1, v_2, \dots, v_M\}$.

- The state transition probability matrix $A = \{a_{ij}\}$, where a_{ij} is the probability of moving from state S_i to state S_j: $a_{ij} = \text{Prob}[q_{t+1} = S_j \mid q_t = S_i]$ for $1 \leq i \leq N$ and $1 \leq j \leq N$, with proper normalization $\forall_{1 \leq i \leq N} \sum_j a_{ij} = 1$.

- The emission probabilities $B = \{b_j(k)\}$ in state S_j, where $b_j(k)$ is the probability of emitting symbol v_k at state S_j: $b_j(k) = \text{Prob}[v_k \text{ at } t \,|\, q_t = S_j]$ for $1 \leq j \leq N$ and $1 \leq k \leq M$.

- The initial state distribution $\pi = \{\pi_i\}$, where π_i is the probability that the start state is S_i: $\pi_i = \text{Prob}[q_1 = S_i]$ for $1 \leq i \leq N$.

In typical bioinformatics applications two related problems are fundamental:

Problem 1. Given a Hidden Markov Model λ and an observed sequence $O = o_1 o_2 \ldots o_T$, compute the probability $\text{Prob}[O \,|\, \lambda]$ that the HMM can generate the sequence. This value indicates the significance of the observation. The problem is commonly solved by applying the *forward algorithm.*

Problem 2. Given a Hidden Markov Model λ and an observed sequence $O = o_1 o_2 \ldots o_T$, choose a corresponding state path $Q = q_1 q_2 \ldots q_T$ which best "explains" the observations, i.e., which maximizes $\text{Prob}[Q \,|\, O, \lambda]$. This problem can be solved by the *Viterbi algorithm* [78].

8.3 Secure Forward Algorithm

We consider the following two-party scenario. Party $\mathcal{A}$ knows a genomic sequence O, while party $\mathcal{B}$ commands over a specific HMM λ. $\mathcal{A}$ could be a health care provider, who has sequenced a patient's genome O, but wishes to preserve the privacy of the patient. $\mathcal{B}$ is a drug company or some bioinformatics institute, which wants to preserve its intellectual property contained within the parameterization of the HMM. Both parties are interested to learn how good the model fits to the genome O by running the forward algorithm, whereas neither party wants to share its input with each other. The overall probability reported by the algorithm can be, for example, the confidence of $\mathcal{B}$ that $\mathcal{A}$'s patient develops a particular disease.

To compute the probability $\text{Prob}[O \,|\, \lambda]$ for some sequence $O = o_1 o_2 \ldots o_T$ of length T, we can employ the forward algorithm. Consider the so-called forward variable

$$\alpha_t(i) = \text{Prob}[o_1, \ldots, o_t, q_t = S_i \,|\, \lambda]$$

which indicates how likely it is to see the sequence $o_1, \ldots, o_t$ and to end up in state S_i after processing t steps. For $\alpha_1(i)$ we have $\alpha_1(i) = \pi_i b_i(o_i)$. Given

Protocol 17 Secure Forward Algorithm (Part 1)

```
Input: Party $\mathcal{A}$: Sequence $O = o_1 o_2 \dots o_T$
       Party $\mathcal{B}$: HMM $\lambda = (A, B, \pi)$
Output: $\text{Prob}[O \mid \lambda]$
1: Initialization:
   Party $\mathcal{A}$: For each $o_i$ prepare a vector $\Theta_i = \{\theta_{i1}, \dots, \theta_{iM}\}$ in a way that
   $\theta_{ij} = 1$ if $v_j = o_i$ and $\theta_{ij} = 0$ otherwise.
   Encrypt $\Theta_i$ component wise and send $[\Theta_i]$ to $\mathcal{B}$
2: Party $\mathcal{B}$: // Compute emission probabilities:
   for $i = 1$ to $N$
      for $j = 1$ to $T$
         $[\rho_{b_i(o_j)}] = \prod_{k=1}^{M} [\theta_{ik}]^{\rho_{b_i(v_k)}}$
         $[\tau_{b_i(o_j)}] = \prod_{k=1}^{M} [\theta_{ik}]^{\tau_{b_i(v_k)}}$
         $[b_i(o_j)] := ([\rho_{b_i(o_j)}], [\tau_{b_i(o_j)}])$
      end
   end
3: Party $\mathcal{B}$:
   for $i = 1$ to $N$
      $[\alpha_1(i)] = \mathbf{LPROD}([\pi_i], [b_i(o_1)])$
   end
```

$\alpha_t(i)$, the probabilities $\alpha_{t+1}(i)$ can be computed inductively by

$$\alpha_{t+1}(i) = b_i(o_{t+1}) \cdot \sum_{j=1}^{N} \alpha_t(j) \cdot a_{ji}.$$

Finally, we have

$$\text{Prob}[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i).$$

A full description of the realization of the forward algorithm using the framework of Section 5.2 can be found in Protocols 17 and 18. Note that in the protocols for clearness of presentation we omit to explicitly label encoded elements: all values $a_{ij}, b_i(v_j), \pi_i$ and α_i should be read as encoded values according to Section 5.2.

In the initialization step, party $\mathcal{A}$ provides party $\mathcal{B}$ with an encrypted version of the sequence, where each symbol of O is encoded as a binary vector of length M (the position of the one in the vector encodes the symbol). This allows party

Protocol 18 Secure Forward Algorithm (Part 2)

```
4: Induction:
   for $t = 1$ to $T - 1$
      for $i = 1$ to $N$
         $[\Sigma_1^1] = \mathbf{LPROD}([\alpha_t(1)], [a_{1i}])$
         for $j = 2$ to $N$
            $[\Sigma_1^j] = \mathbf{LSUM}([\Sigma_1^{j-1}], \mathbf{LPROD}([\alpha_t(j)], [a_{ji}]))$
         end
         $[\alpha_{t+1}(i)] = \mathbf{LPROD}([b_i(o_{t+1})], [\Sigma_1^N])$
      end
   end
5: Termination:
   $[\Sigma_1^2] = \mathbf{LSUM}([\alpha_T(1)], [\alpha_T(2)])$
   for $i = 3$ to $N$
      $\mathbf{LSUM}([\Sigma_1^{i-1}], [\alpha_T(i)])$
   end
   return $[\mathrm{Prob}[O \mid \lambda]] := [\Sigma_1^N]$
```

$\mathcal{B}$ to easily compute encryptions $[b_i(o_j)]$ of the emission probabilities by using the homomorphic properties of the cryptosystem in step 2. In addition, party $\mathcal{B}$ initializes the values $\alpha_1(i)$ as the product of the probabilities π_i and the emission probabilities $b_i(o_1)$ for the first observation symbol (step 3). In step 4 the parties compute interactively the forward-variables $\alpha_t(i)$, and in step 5 the result of the forward algorithm $[\mathrm{Prob}[O \mid \lambda]]$, which can be decrypted by $\mathcal{A}$.

Concerning complexity, note that the induction in step 3 is iterated $T-1$ times and in each iteration the protocols **LPROD** and **LSUM** are used approximately N^2 times each. Thus the forward algorithm requires $\mathcal{O}(N^2 \cdot T)$ **LPROD** and **LSUM** operations.

8.4 Secure Viterbi Algorithm

Returning to our previous example of a health care provider $\mathcal{A}$ and a bioinformatics company $\mathcal{B}$, the two parties might face an additional problem: given a protein or a genomic sequence O of $\mathcal{A}$'s patient, where exactly are signals of malicious mutations or other sequence changes, that might eventually be targeted by gene therapy? Here $\mathcal{B}$ needs to report the best matching state path Q back

Protocol 19 Secure Viterbi Algorithm (Part 1)

```
Input: Party $\mathcal{A}$: Sequence $O = o_1 o_2 \dots o_T$
    Party $\mathcal{B}$: HMM $\lambda = (A, B, \pi)$
Output: Party $\mathcal{B}$ state sequence $q_1 q_2 \dots q_t$
    Initialization:
1-2: These steps are identical to steps 1-2 of the forward algorithm
  3: Party $\mathcal{B}$:
    for $i = 1$ to $N$
      $[\delta_1(i)] = \mathbf{LPROD}([\pi_i], [b_i(o_1)])$
    end
  4: Induction:
    for $t = 2$ to $T$
      for $i = 1$ to $N$
        $[\mathbb{M}_1^1] = \mathbf{LPROD}([\delta_{t-1}(1)], [a_{1i}])$
        $[\Psi_1^1] = [1]$
        for $j = 2$ to $N$
          $[tmp] = \mathbf{LPROD}([\delta_{t-1}(j)], [a_{ji}])$
          $\gamma_1 := ([\mathbb{M}]_1^{j-1}, [\Psi_1^{j-1}])$
          $\gamma_2 := ([tmp], [j])$
          $([\mathbb{M}_1^j], [\Psi_1^j]) = \mathbf{LMAX}(\gamma_1, \gamma_2)$
        end
        $[\delta_t(i)] = \mathbf{LPROD}([b_i(o_t)], [\mathbb{M}_1^N])$
        $[\psi_t(i)] := [\Psi_1^N]$
      end
    end
```

to $\mathcal{A}$, in a way that $\mathcal{A}$ can identify certain regions, e.g. protein-DNA binding domains, that might be malicious, thus excluding "healthy" regions from further consideration and treatment. Again, $\mathcal{A}$ has to protect the privacy of the patient by not disclosing the sequence O, and $\mathcal{B}$ does not want to report details of the topology and parameterization of the HMM because of intellectual property or other business related reasons. In order to interpret the state sequence Q, $\mathcal{A}$ only needs information on how to interpret special states that indicate diseases, while the rest of the HMM (in particular the parameterization of the probabilistic transitions) can remain hidden.

The Viterbi algorithm finds a state path $Q = q_1 q_2 \dots q_T$ that best explains the observation $O = o_1 o_2 \dots o_T$, i.e., which maximizes $\text{Prob}[Q \,|\, O, \lambda]$. To find such a state sequence Q for a given observation sequence O, we compute probabilities

$$\delta_t(i) = \max_{q_1 q_2 \cdots q_{t-1}} \text{Prob}[q_1 q_2 \ldots q_t = i, o_1 o_2 \ldots o_t \mid \lambda],$$

where $\delta_t(i)$ is the highest likelihood of all paths of length t (derived from the first t input symbols $o_1, \ldots, o_t$) which end in state S_i. Given $\delta_t(i)$, the value $\delta_{t+1}(i)$ can be computed inductively by

$$\delta_{t+1}(i) = b_i(o_{t+1}) \cdot \max_{1 \leq j \leq N} \delta_t(j) \cdot a_{ji}.$$

In addition to $\delta_t(i)$ the Viterbi Algorithm makes use of another variable $\psi_{t+1}(i)$, which allows keeping track of the index of the chosen maximum:

$$\psi_{t+1}(i) = \text{argmax}_{1 \leq j \leq N} \delta_t(j) \cdot a_{ji}.$$

When all symbols from the input sequence are processed, the algorithm performs a backtracking step to compute the state sequence $Q = q_1 q_2 \ldots q_T$ by using the values $\psi_t(i)$. We set q_T to the state which maximizes $\psi_T(i)$ for $1 \leq i \leq N$. Since $\psi_t(i)$ records the best matching predecessor of each state, we can compute the optimal state sequence in reverse order by $q_T, \psi_T(q_T), \psi_{T-1}(\psi_T(q_T))$, $\ldots, \psi_2(\psi_3(\ldots \psi_T(q_T)))$.

Protocols 19 and 20 depict a secure realization of the Viterbi algorithm. Using the same notation as in the forward algorithm, all values $a_{ij}, b_i(v_j), \pi_i$ and α_i should be read as encoded triplets according to Section 5.2. The protocol starts in a similar way as Protocol 17, namely by providing party $\mathcal{B}$ with encryptions of the emission probabilities $b_i(o_j)$. In step 3 party $\mathcal{B}$ initializes the values $\delta_1(i)$. In the induction step the parties interactively compute the maximum of the values **LPROD**$([\delta_{t-1}(j)], [a_{ji}])$, where $1 \leq j \leq N$. Along with the maximum they compute for each state an encryption of $[\psi_t(i)]$. In steps 6 and 7 they perform backtracking, providing party $\mathcal{A}$ with the most likely path $q_1 q_2 \ldots q_T$. This is done as follows: For each $2 \leq t \leq T$, starting with $t = T$, party $\mathcal{B}$ computes values $[v_i] = [(v - i) \cdot r_i + \psi_{t-1}(i)]$, where v is the current (best matching) state q_t. Thus the value $(v - i)$ will be equal to 0 if and only if $i = q_t$. In this case $(v - i) \cdot r_i$ will be equal to 0 and therefore $[v_i] = [(v - i) \cdot r_i + \psi_{t-1}(i)]$ contains the correct predecessor state $q_{t-1} = \psi_{t-1}(q_t)$. In any other case, $(v - i) \cdot r_i$ will evaluate to a random value from $\mathbb{Z}_u$, which will be, with very high probability, greater than N (the number of possible states).

In terms of complexity, the Viterbi and forward algorithm are very similar: each **LSUM** operation in the forward algorithm is replaced by **LMAX**; furthermore, the Viterbi algorithm requires backtracking to compute the final result. In

Protocol 20 Secure Viterbi Algorithm (Part 2)

```
5: Termination:
   $([\mathbb{M}_1^2], [\Psi_1^2]) = \mathbf{LMAX}(([\delta_T(1)], [1]), ([\delta_T(2)], [2]))$
   for $i = 3$ to $N$
      $([\mathbb{M}_1^i], [\Psi_1^i]) = \mathbf{LMAX}(([\mathbb{M}_1^{i-1}], [\Psi_1^{i-1}]), ([\delta_T(i)], [i]))$
   end
   $[q_T] := [\Psi_1^N]$
   Send $q_T$ to party $\mathcal{A}$
   Set $t := T + 1$, $[v'] = [q_T]$
6: Backtracking
   Party $\mathcal{B}$:
   Set $t := t - 1$, $[v] := [v']$
   If $t = 1$ terminate, else:
   Choose random values $r_1, \ldots, r_N \in_R \mathbb{Z}_u$
   for $i = 1$ to $N$
      $[v_i] = [(v - i) \cdot r_i + \psi_{t-1}(i)]$
   end
   Send $[v_1], \ldots, [v_N]$ to Party $\mathcal{A}$
7: Party $\mathcal{A}$:
   $v_i = \text{Decrypt}([v_i])$ for $1 \leq i \leq N$
   Find value $v' \in \{v_1, \ldots, v_N\}$ for which it holds $1 \leq v' \leq N$
   $q_t := v'$
   Send $[v']$ to party $\mathcal{A}$
   Go back to step 6
```

summary, the Viterbi algorithm requires $\mathcal{O}(TN^2)$ **LPROD** and **LMAX** operations.

8.5 Implementation and Experimental Results

We have implemented the framework for performing secure computations on non-integer values as well as the algorithms to evaluate HMMs in C++ using the GNU GMP library version 5.0.1. As hash function we used the SHA-2 implementation included in Crypto++ Library 5.6.1 [79]. Tests were run on a computer with a 2.1 GHz 8 core processor and 4GB of RAM running Ubuntu version 9.04. The parties $\mathcal{A}$ and $\mathcal{B}$ were implemented as two multi-threaded entities of the same

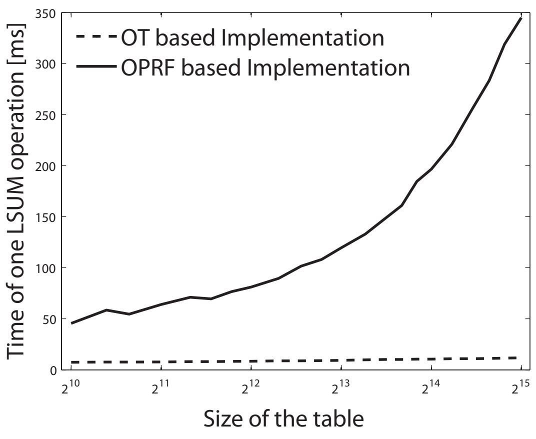

Figure 8.1: Performance of the **LSUM** operation.

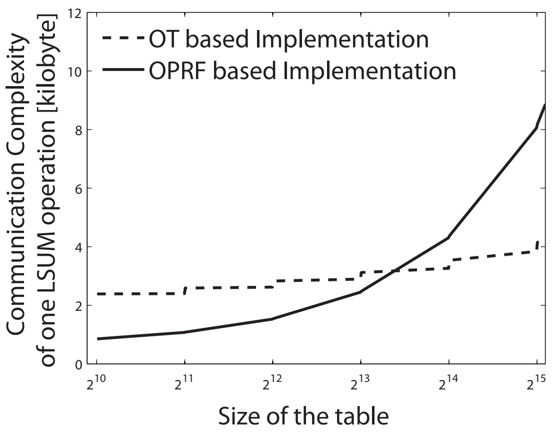

Figure 8.2: Communication complexity of the **LSUM** operation.

program. Therefore our tests do not include network latency (however, many of the computations can be run in parallel and do not depend on each other, thus the different threads could continue computing while sending and receiving messages).

8.5.1 Complexity of LSUM

Since computing the sum of two encoded elements is the most complex basic operation, we first focus on an analysis of **LSUM**. We have implemented two versions of the **LSUM** operation: The first implementation uses the OPRF construction from Section 5.1.1 based on the Paillier cryptosystem. The second implementation builds on the OT based construction presented in Section 5.1.2, using the DGK cryptosystem. Both versions of the **LSUM** have been implemented with all possible optimizations, in particular, both implementations only operate on positive values, neglecting computations for zero- and sign- flags. The RSA moduli for both Paillier and DGK were chosen to have 1024 bit, while the size of one garbled circuit key was set to 80 bits. We compare the results of both implementations. Figure 8.1 depicts the computational complexity. Both programs were run for different table sizes $|\mathcal{T}|$ in the range between $|\mathcal{T}| = 2^{10}$ and $|\mathcal{T}| = 2^{15}$, each implementation was allowed to reuse each table (in the OPRF construction) resp. each set of polynomials (in the OT based construction) 50 times. Figure 8.1 depicts the results: The Y-axis shows the computational complexity in milliseconds (wall clock time) required to perform one **LSUM**, for different table sizes (X-axis, logarithmic scale). While the OPRF based construction grows quickly in $|\mathcal{T}|$, it can be seen that our OT based solution only grows moderately from 7.3ms for $|\mathcal{T}| = 2^{10}$ to 11.7ms for $|\mathcal{T}| = 2^{15}$. The large difference in the measured timings can be explained by the fact that the complexity of the OPRF based construction is dominated by the construction of the table $\mathcal{T}$, thus the amount of work grows linearly with $|\mathcal{T}|$. The amount of work in the OT based construction is dominated by constructing and evaluating the garbled circuits in Step 2 of Protocol 7, which grows only with $\log(|\mathcal{T}|)$.

Figure 8.2 depicts the amortized communication complexity of a single **LSUM** operation. The Y-axis plots the communication in kilobytes. For the OPRF based construction, this is dominated by transmitting the table $\mathcal{T}$. In the OT based construction, again the transmission of the garbled circuits dominates the communication complexity. We see that the communication complexity of the OPRF based construction indeed scales perfectly linear with the table size, while the complexity of the OT based construction is dominated by the logarithmic growth of the garbled circuits.

Thus, comparing the two implementations, we see that the OT based construction has a very low computational complexity compared to the OPRF based construction. In terms of communication, the OPRF based construction is about five times more efficient for small table sizes $|\mathcal{T}|$. According to the specific equipment and bandwidth limitations, one might choose one or the other solution, or a combined approach where a certain percentage of the **LSUM** operations is computed using the OT based solution, and the rest using the OPRF based construction. In the remaining tests we will use the OT based construction.

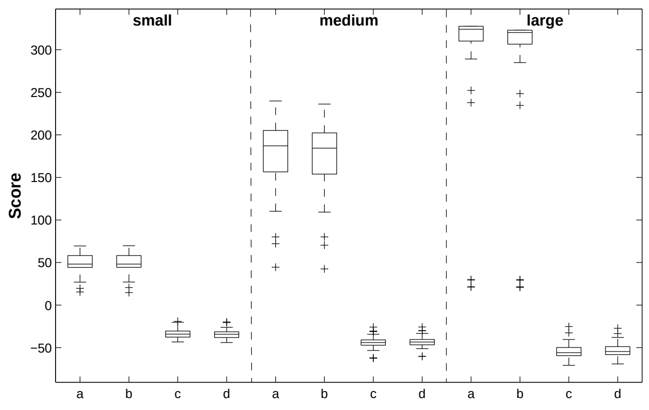

Figure 8.3: Boxplots of the scores of HMMER and our implementation.

8.5.2 Complexity of private HMM analysis.

In order to demonstrate the practicality of the secure forward and the Viterbi algorithm, we implemented the forward and Viterbi algorithms in HMMER [80], version 2.3.2, which is widely used in the bioinformatics community to perform analysis of DNA and protein sequences. Real values were encoded and encrypted as described in Section 5.2, while the **LSUM** operation was realized by using the OT based construction implemented for positive values.

HMMs used in HMMER. HMMER uses a special class of HMMs, namely Profile Hidden Markov Models (PHMMs) [81], which have a specific topological structure; in particular, each state in the model can have at most $T \ll N$ successors (i.e., all other transitions are implicitly set to zero). Since the general structure of such models is publicly known, this allows to decrease the asymptotic complexity of both algorithms to $\mathcal{O}(TN)$. In order to be able to compare the accuracy of our results with HMMER, we decided to implement the same variant of forward and Viterbi algorithm that is used in HMMER.

We tested our implementation with several HMMs from the PFAM database [82]. Among them are models which are most relevant for identification of protein domains, that play crucial roles in oncogenic transformations and other diseases connected to signaling in cells. In particular, we chose HMMs of three different sizes: A small sized model (SH3_1, PF00018) of length 48, a medium sized model (Ras, PF00071) of length 162 and a large model (BID, PF06393) of length 196. Here length refers to the number of sites N.

Parameter Choices for Number Representation. Through empirical tests on a large set of candidate parameters, we determined B, S, C and the table size $|\mathcal{T}|$ in a way that our secure realization of the HMM algorithms can differentiate well between matching and non-matching sequences; in particular we chose the parameters from the candidate set that yielded the smallest deviation from the HMMER scores and limited the necessary table size.

We fixed parameters (see Section 5.2) $B = 2.71$, $S = 10$, while $\log_2 C$ is set to 100, 250 and 330 for the small, medium and large models. Note that the intermediate results during the computation can become quite large (in particular, this is the case when a genomic sequence matches well against the model) and can lie in the range of $[0; 2^{200}]$ and even above in rare cases. We further note that all log-odd score values are positive. For the small, medium and large models we chose tables of size 1400, 3520 and 4600, respectively, in the **LSUM** operation.

Despite heavy quantization, our solution is still numerically accurate: In particular, we compared the scores of the forward algorithm, as produced by HMMER, with the scores of our implementation for 50 matching and 50 non-matching sequences. Figure 8.3 shows (for the small, medium and large model) four boxplots: The scores of the original HMMER implementation (a) and our privacy preserving implementation (b) for matching sequences; and the scores of the HMMER implementation (c) and our implementation (d) for non-matching sequences. It can be seen that corresponding boxes (a and b, as well as c and d) show a close to perfect overlap; thus, despite the quantization inflicted by the logarithmic representation, our implementation yields very similar scores as HMMER. Furthermore, the errorbars of the boxplots b and d do not overlap, which shows that our privacy perserving implementation can perfectly distinguish matching from non-matching sequences. Thus, the number representation is suitable for the problem domain.

Algorithm:	**LSUM (LMAX)**	**LPROD**
Forward algorithm (Viterbi)		
- small model	14,161	25,971
- medium model	162,491	297,929
- large model	225,984	414,337

Table 8.1: Operation count for Forward- and Viterbi algorithm.

Algorithm	Small	Medium	Large
Forward	22	298	449
Viterbi	94	933	1357

Table 8.2: Computational complexity in seconds.

8.5.2.1 Computational complexity

We measure the computational complexity of the forward Algorithm (Protocols 17 and 18) and the Viterbi algorithm (Protocols 19 and 20). Table 8.2 depicts the average runtime (wall clock time) of a single matching operation. On the average, 14,161, 162,491 and 225,984 **LSUM** operations had to be performed for the small, medium and large model when applying the forward algorithm for various different test sequences of different length. In the same tests, the average count for **LPROD** operations were 25,971 for the small, 297,929 for the medium and 414,337 for the large model. Note that in these tests we allowed parallel computations on multiple cores of the processor (e.g., we allowed to parallelize computation of the polynomials, run multiple **LSUM** operations in parallel, etc). Even though we did not fully optimize our programs at this point (only 3 out of 8 cores were used on average), we believe that the results are nevertheless insightful: For example, running the forward algorithm on the medium sized model requires approximately 5 minutes, despite performing 297,929 invocations of **LPROD** and 162,491 invocations of **LSUM**.

Algorithm	Small	Medium	Large
Forward	182.0	844.0	1222.8
Viterbi	60.0	701.4	1030.0

Table 8.3: Communication complexity in MB.

8.5.2.2 Communication complexity

The communication complexity measures the traffic between the two parties. This value mainly depends on the size of the RSA modulus n, which was set to 1024 bits. The key size for the garbled circuits was set to 80 bit.

For the forward algorithm, the major part of the communication complexity is caused by transmitting the tables used for the **LSUM** operations. For the Viterbi algorithm, communication complexity depends mainly on the underlying comparison protocol **LMAX**. Table 8.3 depicts the communication complexity for both protocols.

Chapter 9

Conclusions and Future Work

In this thesis we presented different approaches which allow to compute with non-integer values in the encrypted domain. We have shown that it is possible, and in fact practical to perform these computations. In particular, we investigated two different approaches on how to represent and compute with values taken from a real domain and provided a theoretical and practical evaluation of their properties:

Secure computations using logarithmic encoding. The first approach represents values using a logarithmic encoding. This yields to an representation scheme with constant relative error which can be specified in an easy and accurate way by adjusting a set of diferent parameters. This approach allows for very efficient arithmetic operations on values encoded and encrypted using the logarithmic representation. In particular, it resulted in a computational framework that allows to process both very small and very large values with good practical performance. We provided secure two-party protocols to perform all arithmetic operations on encrypted and encoded values in an oblivious way.

Securely processing encrypted floating point values. The second approach implements the well known IEEE 754 floating point standard in a two party setting. Our secure implementation strictly follows the IEEE 754 standard and stores sign, mantissa and exponent as different encryptions. Our protocols allow two parties $\mathcal{A}$ and $\mathcal{B}$ to privately perform all arithmetic computations on floating point values. These protocols can be seen as an initial solution to this problem. While many further optimizations are possible, we believe that our protocols can be seen as basis for many practical applications. While the performance of the arithmetic operations is slightly worse than for logarithmic encodings for small applications, this approach has a better asymptotic behavior and is therefore better suited for large applications.

As future work, we suggest to investigate a more fine grained exception handling. Also, a next step would be to implement a floating point version which uses the full plaintext space of the Paillier cryptosystem. This would allow for larger significants, thus a normalization would not have to be run after each arithmetic operation.

Evaluation of the presented techniques. We showed that the approach taken in this thesis is indeed practical and can be used to solve real world problems. We demonstrated its practicality by applying the framework using logarithmic encoding to the problem of protecting privacy in genomic sequence analysis. In particular, we showed how to interactively process Hidden Markov Models in an oblivious way by applying either the forward or the Viterbi algorithm. Despite the large number of operations performed during the computation (which requires hundreds of thousands of complex cryptographic operations), the result is numerically accurate and can be obtained on standard computing equipment within reasonable time.

Future directions. We see this work as a solid foundation for future work in the field of secure multiparty computation with non-integer values. It can and should be improved further to gain more efficient protocols computing on logarithmic encoding and floating point values; the work presented in this thesis can serve as a reference implementation to identify possible future alternative approaches which might yield to better accuracy or better performance.

We see different ways to continue this line of research in the future. Firstly, one should investigate how to transfer the results presented in this thesis to the setting with more than two parties. As a second direction we suggest to investigate which of the proposed techniques is better suited for parallelization. Since it seems that the current development in computer hardware technology tends to processors with multiple cores instead of one core with faster CPU, it should be desirable to investigate approaches for SMC which can make use of the current development in computer hardware.

References

[1] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, FOCS'1982, 3-5 November 1982, Chicago, Illinois, USA*, pages 160–164. IEEE, 1982.

[2] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC'1987, 25-27 May 1987, New York City, NY, USA*, pages 218–229. ACM, 1987.

[3] Charles Lee and Cynthia C. Morton. Structural genomic variation and personalized medicine. *The New England journal of medicine*, 358(7):740–741, February 2008.

[4] Laura J. van 't Veer and Rene Bernards. Enabling personalized cancer medicine through analysis of gene-expression patterns. *Nature*, 452(7187):564–570, April 2008.

[5] Mike West, Geoffrey S. Ginsburg, Andrew T. Huang, and Joseph R. Nevins. Embracing the complexity of genomic data for personalized medicine. *Genome Research*, 16(5):559–566, 2006.

[6] Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser, and Heike Schröder. Secure computations on non-integer values. In *Proceedings of the Information Forensics and Security (WIFS), 2010 IEEE International Workshop on*, pages 1 –6, dec. 2010.

[7] Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser, and Heike Schröder. Towards secure bioinformatics services. In *Financial Cryptography and Data Security (FC'11)*, 2011.

[8] Martin Franz and Stefan Katzenbeisser. Processing encrypted floating point signals. In *Proceedings of the 13th ACM Workshop on Multimedia and Security (MM&Sec '11)*, MM&Sec '11, New York, NY, USA, 2011. ACM.

[9] Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser, and Heike Schröder. Secure computations on non-integer values with applications to privacy-preserving sequence analysis.

[10] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[11] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC'1982, 5-7 May 1982, San Francisco, California, USA*, pages 365–377. ACM, 1982.

[12] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[13] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.

[14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[15] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 377–394. Springer, 2010.

[16] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology EUROCRYPT 99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin / Heidelberg, 1999.

[17] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[18] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[19] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In Kwangjo Kim, editor, *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC'2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[20] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *Proceedings of the Information Security and Privacy, 12th Australasian Conference, ACISP'2007, Townsville, Australia, July 2-4, 2007, Proceedings*, volume 4586 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2007.

[21] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. A correction to 'efficient and secure comparison for on-line auctions'. *IJACT*, 1(4):323–324, 2009.

[22] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC'2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, 2005.

[23] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *J. Cryptology*, 23(3):422–456, 2010.

[24] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In Reingold [83], pages 577–594.

[25] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2009.

[26] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, 2001.

[27] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.

[28] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[29] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS*, volume 5888 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2009.

[30] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[31] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[32] Sen-Ching S. Cheung and Thinh P. Nguyen. Secure multiparty computation between distrusted networks terminals. *EURASIP J. Information Security*, 2007, 2007.

[33] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *FOCS*, pages 383–395. IEEE, 1985.

[34] Berry Schoenmakers and Pim Tuyls. Efficient binary conversion for paillier encrypted values. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 522–537. Springer, 2006.

[35] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[36] Martin Franz. Privacy-preserving face recognition. *Diplomarbeit, Technische Universität Darmstadt*, 2008.

[37] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In Ian Goldberg and Mikhail J. Atallah, editors, *Privacy Enhancing Technologies*, volume 5672 of *Lecture Notes in Computer Science*, pages 235–253. Springer, 2009.

[38] Markus Jakobsson and Ari Juels. Mix and match: Secure function evaluation via ciphertexts. In Tatsuaki Okamoto, editor, *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2000.

[39] Louis Kruger, Somesh Jha, Eu-Jin Goh, and Dan Boneh. Secure function evaluation with ordered binary decision diagrams. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 410–420. ACM, 2006.

[40] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

[41] Moni Naor and Kobbi Nissim. Communication complexity and secure function evaluation. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(062), 2001.

[42] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Reingold [83], pages 294–314.

[43] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

[44] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[45] VIFF. Virtual ideal functionality framework. http://viff.dk/.

[46] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In Al-Shaer et al. [84], pages 451–462.

[47] Pierre-Alain Fouque, Jacques Stern, and Jan-Geert Wackers. Cryptocomputing with rationals. In Matt Blaze, editor, *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 136–146. Springer, 2002.

[48] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[49] Juan Ramón Troncoso-Pastoriza and Fernando Pérez-González. Efficient protocols for secure adaptive filtering. In *ICASSP*, pages 5860–5863. IEEE, 2011.

[50] Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.

[51] Bart Goethals, Sven Laur, Helger Lipmaa, and Taneli Mielikäinen. On private scalar product computation for privacy-preserving data mining. In Choonsik Park and Seongtaek Chee, editors, *ICISC*, volume 3506 of *Lecture Notes in Computer Science*, pages 104–120. Springer, 2004.

[52] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In Robert Grossman, Roberto J. Bayardo, and Kristin P. Bennett, editors, *KDD*, pages 593–599. ACM, 2005.

[53] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. In Michael Backes and

Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009.

[54] T. Bianchi, A. Piva, and M. Barni. On the implementation of the discrete fourier transform in the encrypted domain. *Information Forensics and Security, IEEE Transactions on*, 4(1):86 –97, march 2009.

[55] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[56] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *J. Cryptology*, 15(3):177–206, 2002.

[57] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[58] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Utku Celik. Privacy preserving error resilient DNA searching through oblivious automata. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 519–528. ACM, 2007.

[59] Ayman Jarrous and Benny Pinkas. Secure hamming distance based computation and its applications. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 107–124, 2009.

[60] P.J.W. Kingsbury, N.G. Rayner. Digital filtering using logarithmic arithmetic. *Electronics Letters*, pages 56–58, 1971.

[61] IEEE. Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[62] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[63] Mikhail J. Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In Sushil Jajodia, Pierangela Samarati, and Paul F. Syverson, editors, *WPES*, pages 39–44. ACM, 2003.

[64] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society, 2008.

[65] Doug Szajda, Michael Pohl, Jason Owen, and Barry G. Lawson. Toward a practical data privacy scheme for a distributed implementation of the smith-waterman genome sequence comparison algorithm. In *NDSS*. The Internet Society, 2006.

[66] Jonathan Katz and Lior Malka. Secure text processing with applications to private DNA matching. In Al-Shaer et al. [84], pages 485–492.

[67] Shu-Yi Su, David J. Balding, and Lachlan J.M. Coin. Disease association tests by inferring ancestral haplotypes using a hidden markov model. *Bioinformatics*, 24(7):972–978, 2008.

[68] Paul D. Thomas and Anish Kejariwal. Coding single-nucleotide polymorphisms associated with complex vs. Mendelian disease: Evolutionary evidence for differences in molecular effects. *Proc. Nat. Acad. Sci.*, 101(43):15398–15403, 2004.

[69] Z. Wei, W. Sun, K. Wang, and H. Hakonarson. Multiple testing in genome-wide association studies via hidden Markov models. *Bioinformatics*, 25(21):2802–2808, 2009.

[70] Hsien-Da Huang, Tzong-Yi Lee, Shih-Wei Tzeng, Li-Cheng Wu, Jorng-Tzong Horng, Ann-Ping Tsou, and Kuan-Tsae Huang. Incorporating hidden markov models for identifying protein kinase-specific phosphorylation sites. *Journal of Computational Chemistry*, 26(10):1032–1041, 2005.

[71] Tim Massingham. Detecting the presence and location of selection in proteins. *Methods Mol Biol*, 452:311–29, Jan 2008.

[72] Nathan O. Stitziel, Yan Yuan Tseng, Dimitri Pervouchine, David Goddeau, Simon Kasif, and Jie Liang. Structural location of disease-associated single-nucleotide polymorphisms. *Journal of Molecular Biology*, 327(5):1021 – 1030, 2003.

[73] Paris Smaragdis and Madhusudana V. S. Shashanka. A framework for secure speech recognition. *IEEE Transactions on Audio, Speech & Language Processing*, 15(4):1404–1413, 2007.

[74] Huseyin Polat, Wenliang Du, Sahin Renckes, and Yusuf Oysal. Private predictions on hidden markov models. *Artificial Intelligence Review*, 34:53–72, 2010. 10.1007/s10462-010-9161-2.

[75] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, pages 267–296, 1990.

[76] J. Baker. The DRAGON system–An overview. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 23, 1975.

[77] Jia Li, Robert M. Gray, and Richard A. Olshen. Joint image compression and classification with vector quantization and a two dimensional hidden markov model. In *Data Compression Conference*, pages 23–32, 1999.

[78] G.D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268 – 278, March 1973.

[79] Crypto++. Library 5.6.1. http://www.cryptopp.com/.

[80] HMMER. Biosequence analysis using profile hidden markov models. http://hmmer.wustl.edu/.

[81] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[82] Pfam. Database version 24.0. http://pfam.sanger.ac.uk.

[83] Omer Reingold, editor. *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*. Springer, 2009.

[84] Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors. *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010.* ACM, 2010.

Wissenschaftlicher Werdegang

Martin Franz

04/2003 – 09/2008	Studium der Mathematik, Technische Universität Darmstadt
10/2003 – 10/2008	Studium der Informatik, Technische Universität Darmstadt
09/2006 – 06/2007	Auslandsaufenthalt an der Universidad de Salamanca (Spanien)
11/2008 – 11/2011	Doktorand, Technische Universität Darmstadt